

Ordenação: algoritmos elementares

Estrutura de Dados — QXD0010



UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

Universidade Federal do Ceará

1º semestre/2020



Introdução



Introdução

- Colocar um vetor numérico em ordem crescente ou decrescente é o primeiro passo na solução de muitos problemas práticos.
- Um vetor pode ser ordenado de muitas maneiras diferentes: algumas elementares, outras mais sofisticadas e eficientes.

Introdução

- Colocar um vetor numérico em ordem crescente ou decrescente é o primeiro passo na solução de muitos problemas práticos.
- Um vetor pode ser ordenado de muitas maneiras diferentes: algumas elementares, outras mais sofisticadas e eficientes.
- Pode-se usar basicamente duas estratégias para ordenar os dados:

- Colocar um vetor numérico em ordem crescente ou decrescente é o primeiro passo na solução de muitos problemas práticos.
- Um vetor pode ser ordenado de muitas maneiras diferentes: algumas elementares, outras mais sofisticadas e eficientes.
- Pode-se usar basicamente duas estratégias para ordenar os dados:
 - inserir os dados na estrutura respeitando sua ordem.
 - a partir de um conjunto de dados já criado, aplicar um algoritmo para ordenar seus elementos.

Ordenação

Um vetor $A[0 \dots n - 1]$ é **crescente** se $A[0] \leq \dots \leq A[n - 1]$.

O problema da ordenação de um vetor consiste no seguinte:

Ordenação

Um vetor $A[0 \dots n - 1]$ é **crescente** se $A[0] \leq \dots \leq A[n - 1]$.

O problema da ordenação de um vetor consiste no seguinte:

- reorganizar (ou seja, permutar) os elementos de um vetor $A[0 \dots n - 1]$ de tal modo que ele se torne crescente.

Ordenação

Um vetor $A[0 \dots n - 1]$ é **crescente** se $A[0] \leq \dots \leq A[n - 1]$.

O problema da ordenação de um vetor consiste no seguinte:

- reorganizar (ou seja, permutar) os elementos de um vetor $A[0 \dots n - 1]$ de tal modo que ele se torne crescente.

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



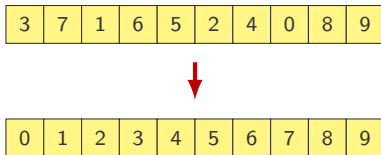
0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

Ordenação

Um vetor $A[0 \dots n - 1]$ é **crescente** se $A[0] \leq \dots \leq A[n - 1]$.

O problema da ordenação de um vetor consiste no seguinte:

- rearranjar (ou seja, permutar) os elementos de um vetor $A[0 \dots n - 1]$ de tal modo que ele se torne crescente.



Nos códigos vamos ordenar vetores de **int**

Ordenação

Um vetor $A[0 \dots n - 1]$ é **crescente** se $A[0] \leq \dots \leq A[n - 1]$.

O problema da ordenação de um vetor consiste no seguinte:

- rearranjar (ou seja, permutar) os elementos de um vetor $A[0 \dots n - 1]$ de tal modo que ele se torne crescente.

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

Um vetor $A[0 \dots n - 1]$ é **crescente** se $A[0] \leq \dots \leq A[n - 1]$.

O problema da ordenação de um vetor consiste no seguinte:

- reorganizar (ou seja, permutar) os elementos de um vetor $A[0 \dots n - 1]$ de tal modo que ele se torne crescente.

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

Um vetor $A[0 \dots n - 1]$ é **crescente** se $A[0] \leq \dots \leq A[n - 1]$.

O problema da ordenação de um vetor consiste no seguinte:

- reorganizar (ou seja, permutar) os elementos de um vetor $A[0 \dots n - 1]$ de tal modo que ele se torne crescente.

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

Um vetor $A[0 \dots n - 1]$ é **crescente** se $A[0] \leq \dots \leq A[n - 1]$.

O problema da ordenação de um vetor consiste no seguinte:

- reorganizar (ou seja, permutar) os elementos de um vetor $A[0 \dots n - 1]$ de tal modo que ele se torne crescente.

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

BubbleSort



BubbleSort – Ordenação por flutuação

Ideia:

BubbleSort – Ordenação por flutuação

Ideia:

- do fim para o começo, vamos trocando pares invertidos

BubbleSort – Ordenação por flutuação

Ideia:

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

BubbleSort – Ordenação por flutuação

Ideia:

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

BubbleSort – Ordenação por flutuação

Ideia:

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

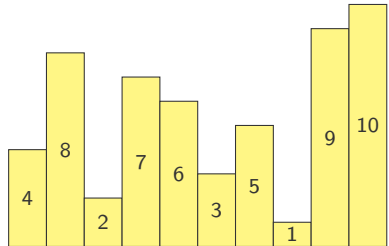
```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```



i

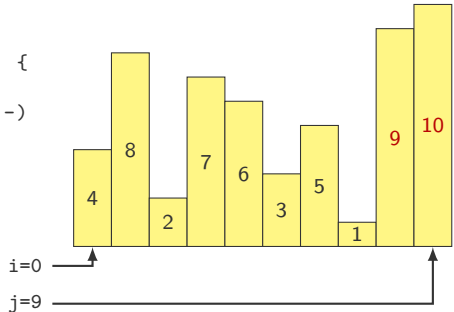
j

BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

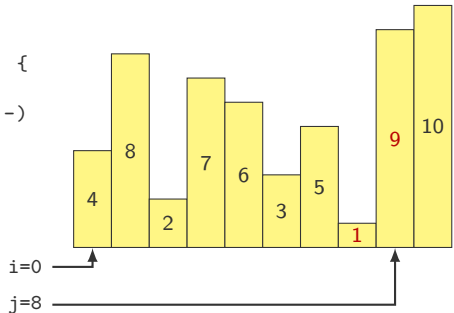


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

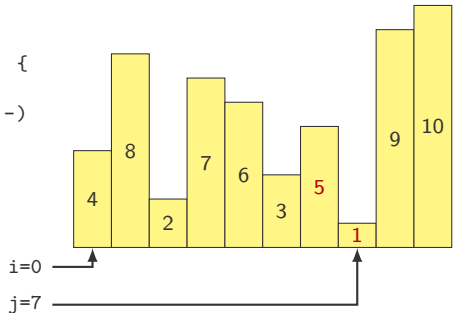


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

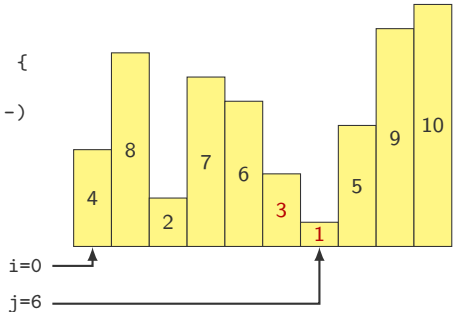


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

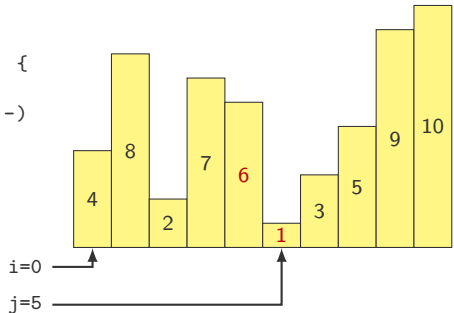


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

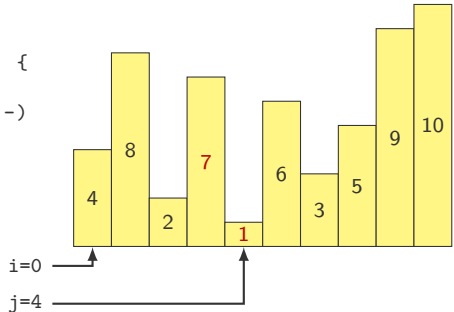


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

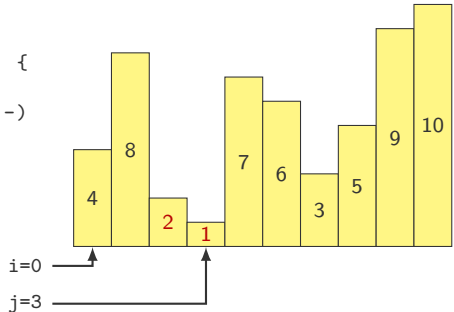


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

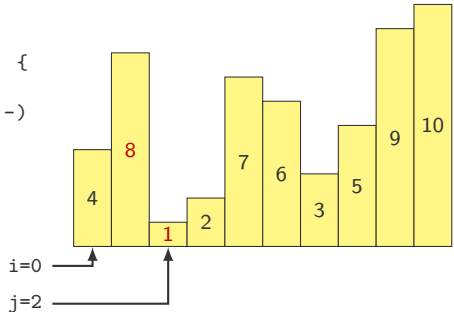


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

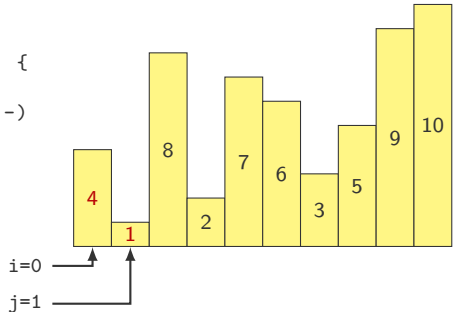


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

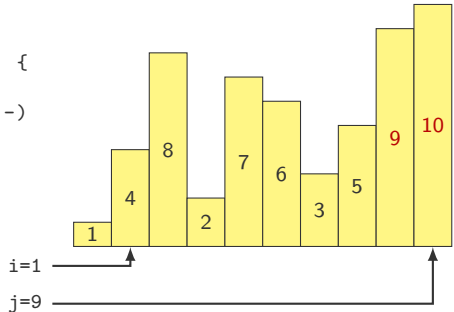


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

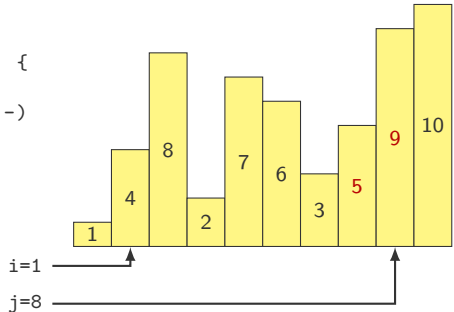


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

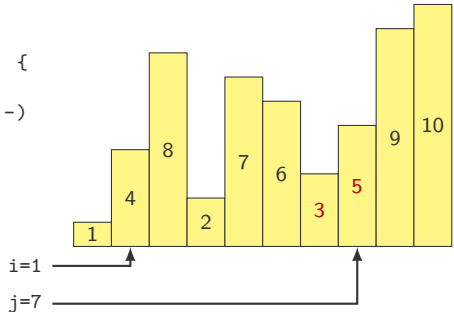


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

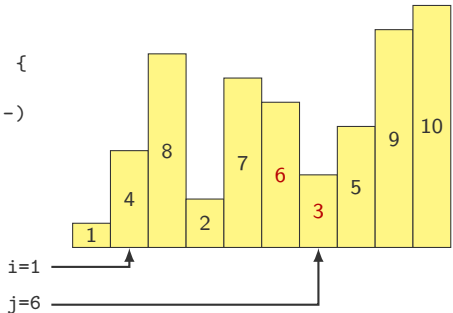


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

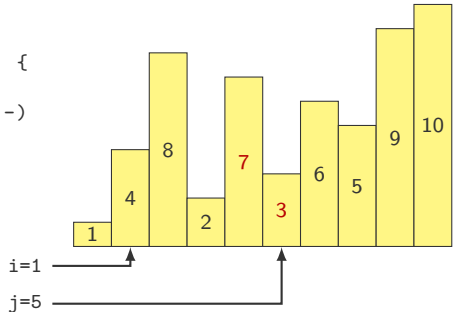


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

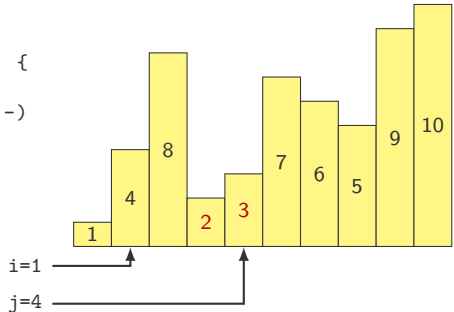


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

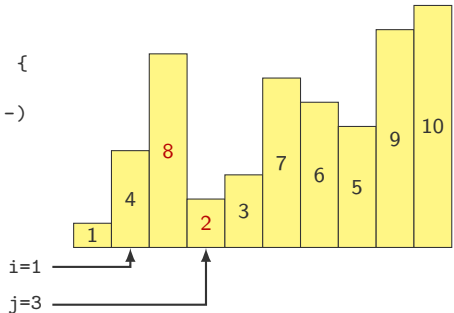


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

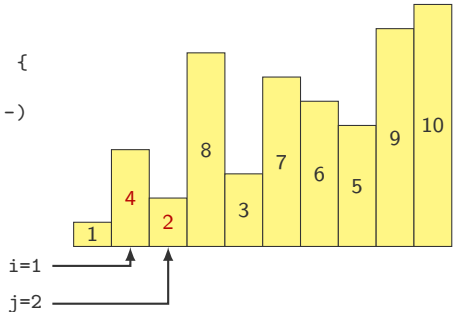


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

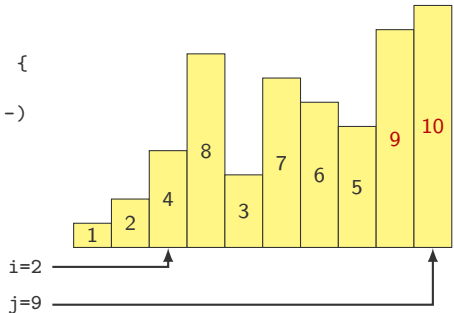


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

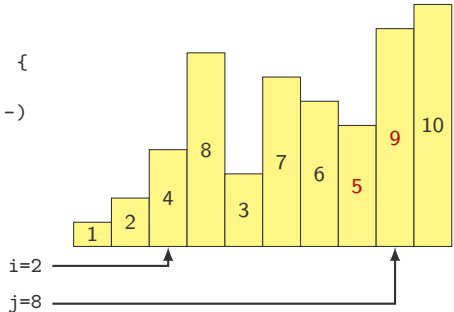


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

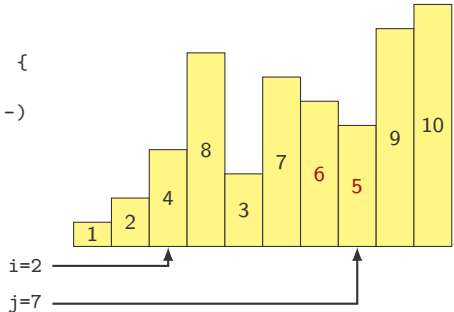


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

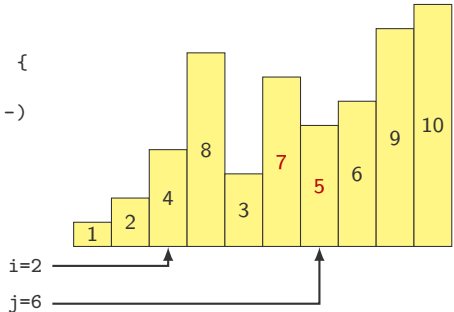


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

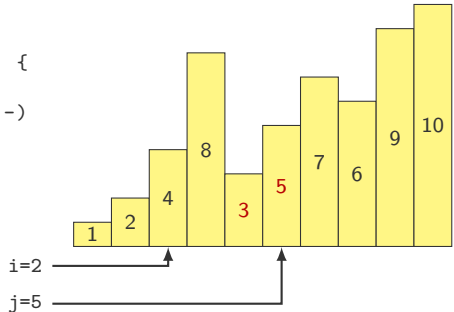


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

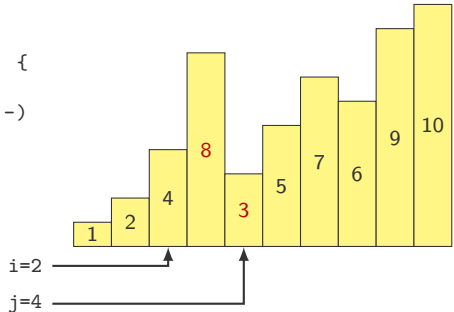


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

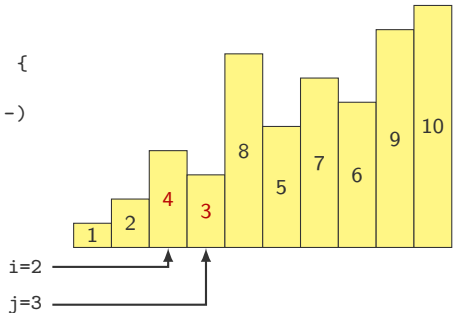


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

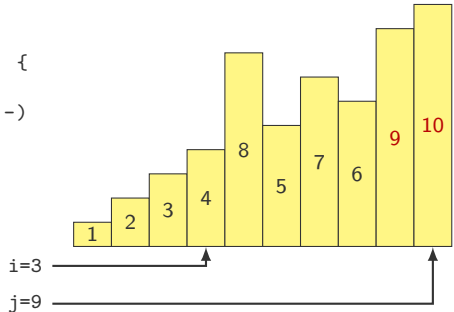


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

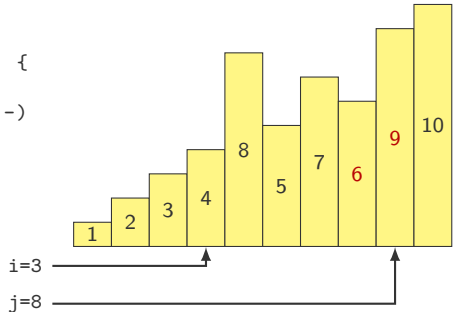


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

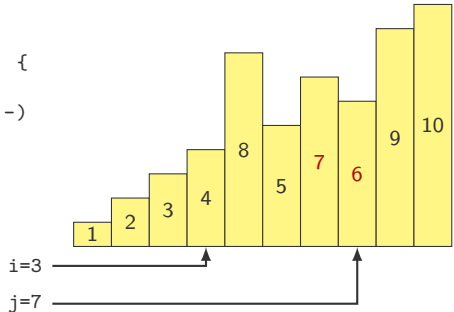


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

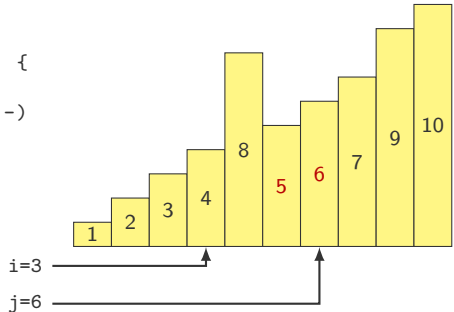


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

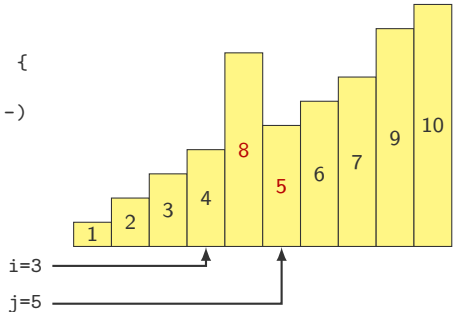


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

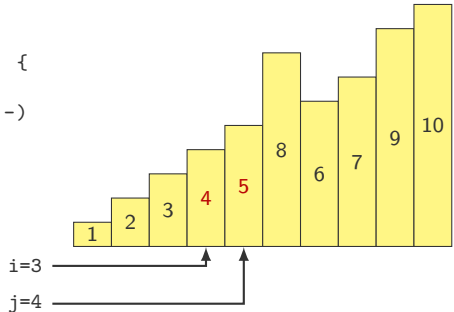


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

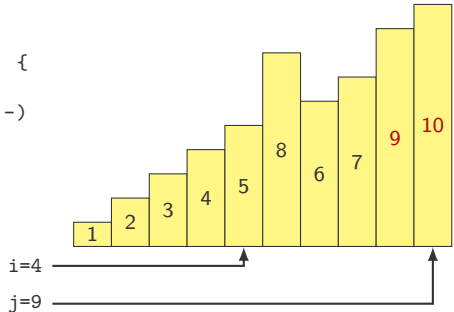


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

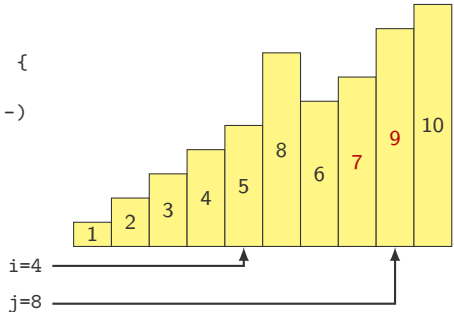


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

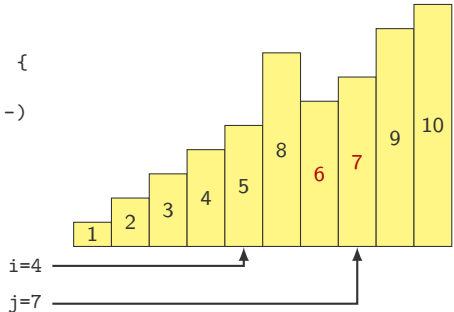


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

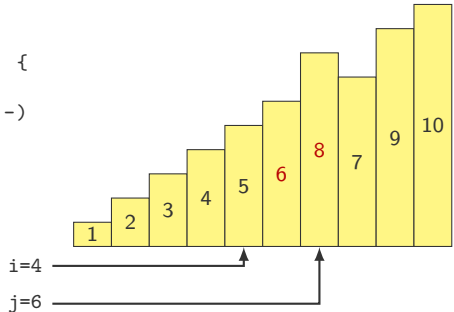


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

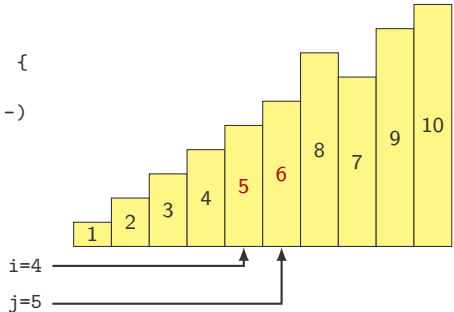


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

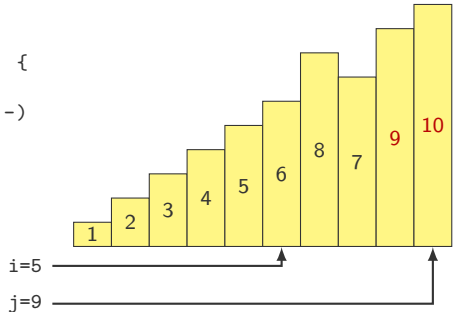


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

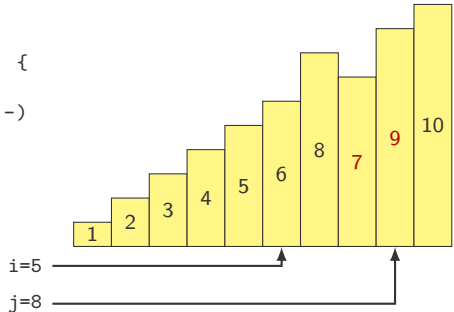


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

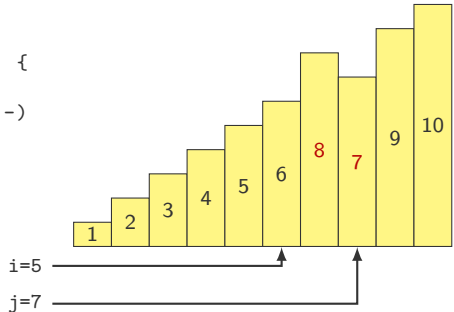


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

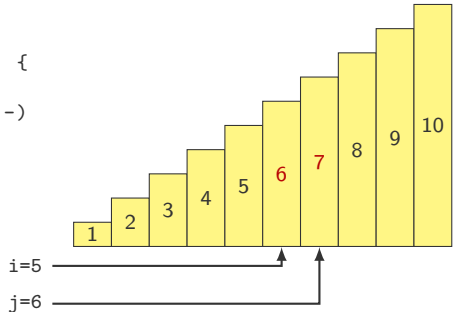


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

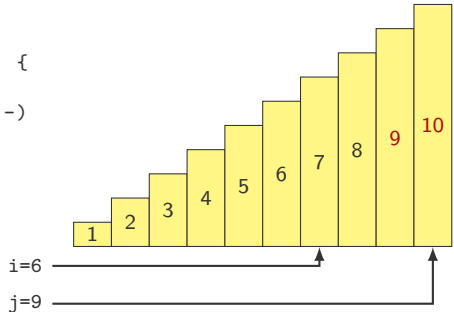


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

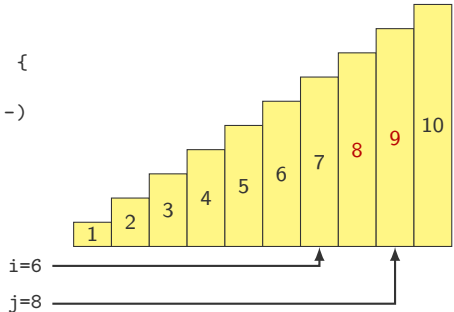


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

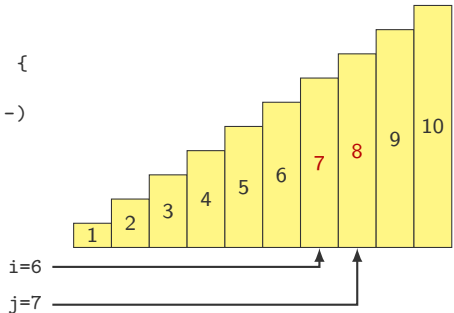


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

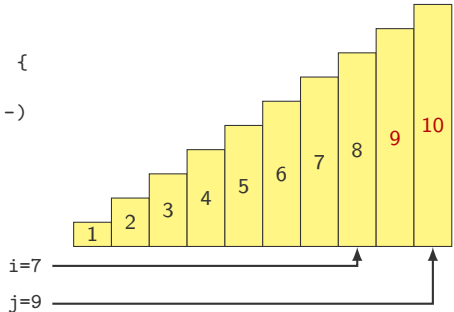


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

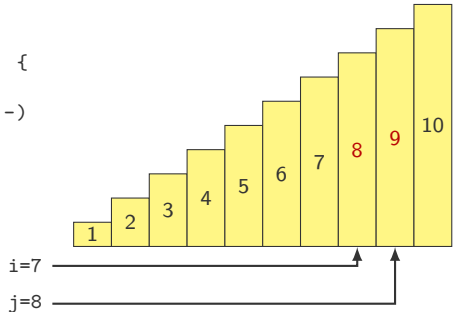


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

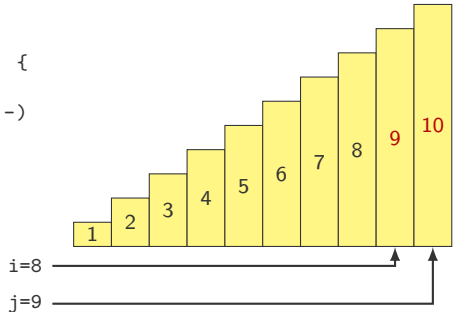


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

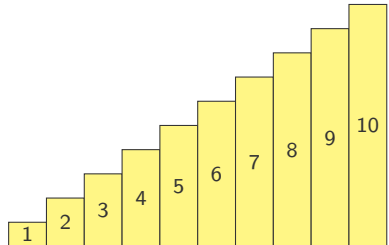


BubbleSort – Ordenação por flutuação

Ideia:

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

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```



i

j

BubbleSort — Complexidade do algoritmo

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

BubbleSort — Complexidade do algoritmo

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

No pior caso toda comparação gera uma troca:

BubbleSort — Complexidade do algoritmo

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

No pior caso toda comparação gera uma troca:

- comparações: $n(n-1)/2 = O(n^2)$

BubbleSort — Complexidade do algoritmo

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

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 — Complexidade do algoritmo

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

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 — Complexidade do algoritmo

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

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 — Complexidade do algoritmo

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

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

BubbleSort — Complexidade do algoritmo

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

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

No melhor caso:

BubbleSort — Complexidade do algoritmo

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

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

No melhor caso:

- comparações: $\approx n^2 = O(n^2)$

Parando quando não há mais trocas

Se não aconteceu nenhuma troca, podemos parar o algoritmo

Parando quando não há mais trocas

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int A[], int n) {  
2     bool trocou = true;  
3     for (int i = 0; i < n-1 && trocou; i++){  
4         trocou = false;  
5         for (int j = n-1; j > i; j--)  
6             if (A[j] < A[j-1]) {  
7                 std::swap(A[j], A[j-1]);  
8                 trocou = true;  
9             }  
10    }  
11 }
```

Parando quando não há mais trocas

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int A[], int n) {  
2     bool trocou = true;  
3     for (int i = 0; i < n-1 && trocou; i++){  
4         trocou = false;  
5         for (int j = n-1; j > i; j--)  
6             if (A[j] < A[j-1]) {  
7                 std::swap(A[j], A[j-1]);  
8                 trocou = true;  
9             }  
10    }  
11 }
```

No pior caso toda comparação gera uma troca:

Parando quando não há mais trocas

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int A[], int n) {  
2     bool trocou = true;  
3     for (int i = 0; i < n-1 && trocou; i++){  
4         trocou = false;  
5         for (int j = n-1; j > i; j--)  
6             if (A[j] < A[j-1]) {  
7                 std::swap(A[j], A[j-1]);  
8                 trocou = true;  
9             }  
10    }  
11 }
```

No pior caso toda comparação gera uma troca:

- comparações: $n(n-1)/2 = O(n^2)$

Parando quando não há mais trocas

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int A[], int n) {
2     bool trocou = true;
3     for (int i = 0; i < n-1 && trocou; i++){
4         trocou = false;
5         for (int j = n-1; j > i; j--){
6             if (A[j] < A[j-1]) {
7                 std::swap(A[j], A[j-1]);
8                 trocou = true;
9             }
10    }
11 }
```

No pior caso toda comparação gera uma troca:

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

Parando quando não há mais trocas

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int A[], int n) {  
2     bool trocou = true;  
3     for (int i = 0; i < n-1 && trocou; i++){  
4         trocou = false;  
5         for (int j = n-1; j > i; j--){  
6             if (A[j] < A[j-1]) {  
7                 std::swap(A[j], A[j-1]);  
8                 trocou = true;  
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 melhor caso:

Parando quando não há mais trocas

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int A[], int n) {  
2     bool trocou = true;  
3     for (int i = 0; i < n-1 && trocou; i++){  
4         trocou = false;  
5         for (int j = n-1; j > i; j--){  
6             if (A[j] < A[j-1]) {  
7                 std::swap(A[j], A[j-1]);  
8                 trocou = true;  
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 melhor caso:

- comparações: $O(n)$

Parando quando não há mais trocas

Se não aconteceu nenhuma troca, podemos parar o algoritmo


```
1 void bubblesort_v2(int A[], int n) {
2     bool trocou = true;
3     for (int i = 0; i < n-1 && trocou; i++){
4         trocou = false;
5         for (int j = n-1; j > i; j--){
6             if (A[j] < A[j-1]) {
7                 std::swap(A[j], A[j-1]);
8                 trocou = true;
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 melhor caso:

- comparações: $O(n)$
- trocas: $O(1)$



InsertionSort



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado

Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta

Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort

Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }
```

Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

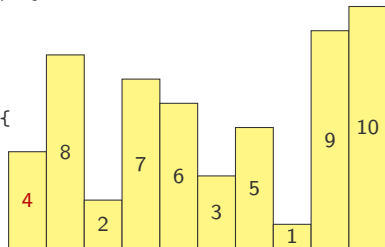
```
1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }
```

Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```

1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }
    
```



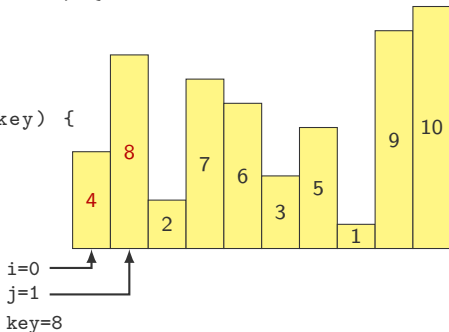
i
j
key

Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```

1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }
    
```

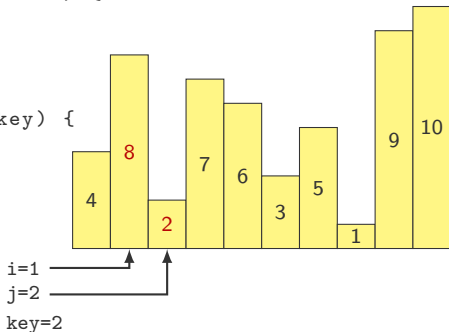


Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```

1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }
    
```

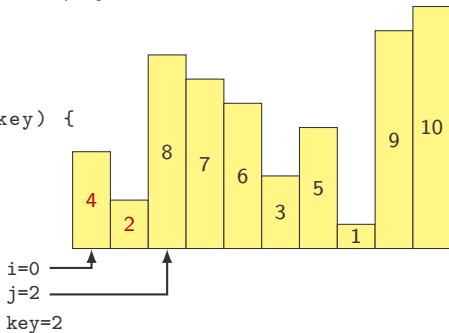


Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```

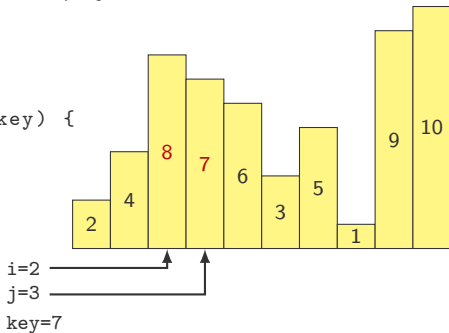
1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }
    
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```

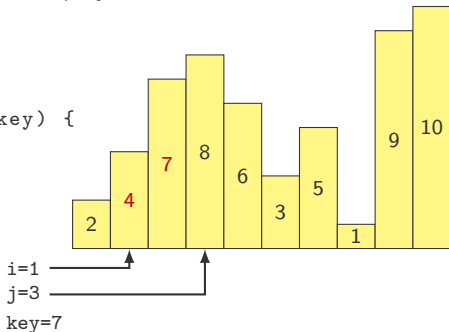


Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```

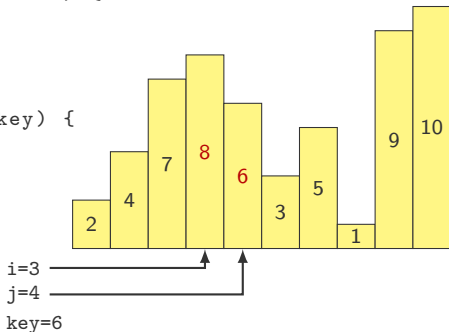
1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }
    
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```

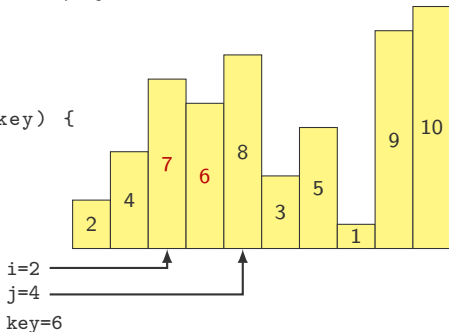


Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```

1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }
    
```

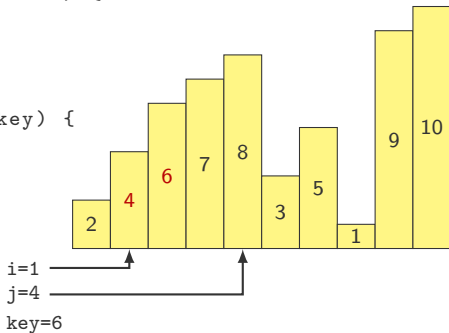


Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```

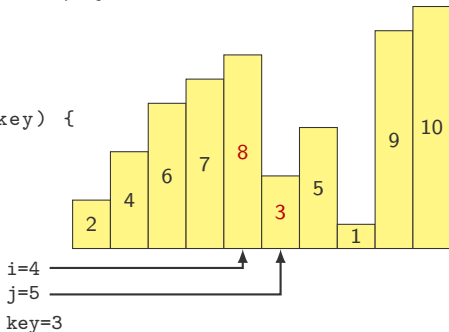
1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }
    
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```

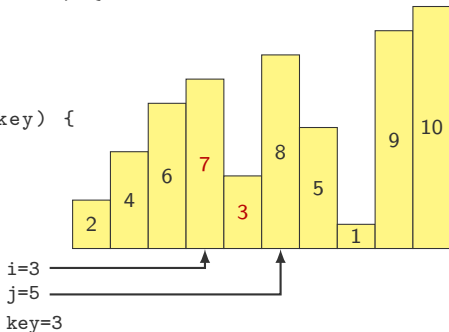


Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```

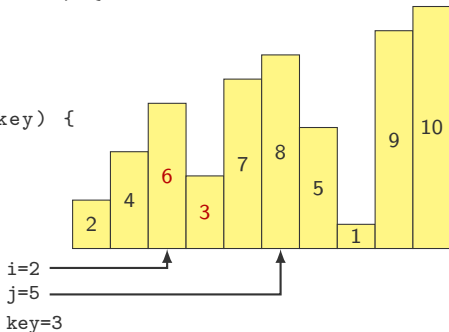
1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }
    
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```

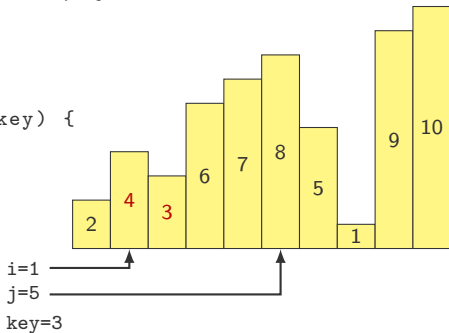


Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```

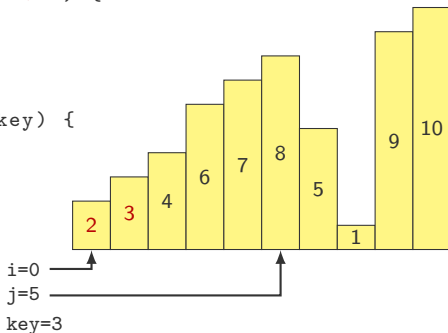
1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }
    
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```



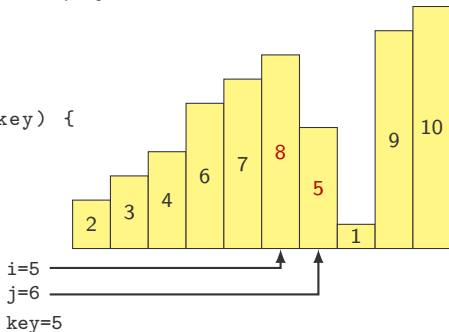
Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```

1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }

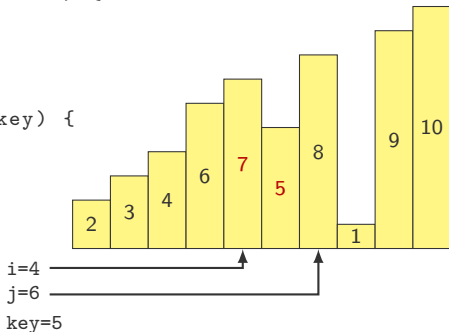
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```

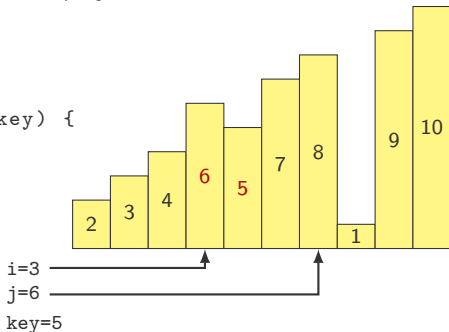


Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```

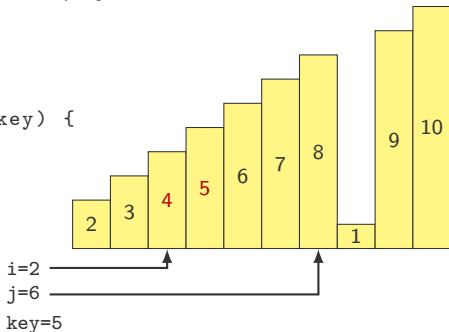
1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }
    
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```

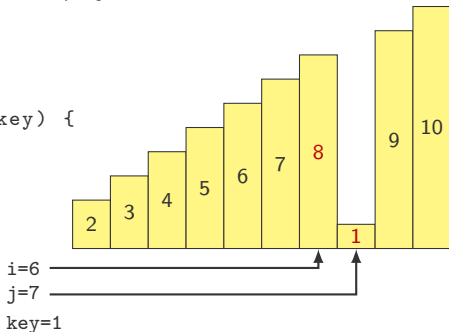


Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```

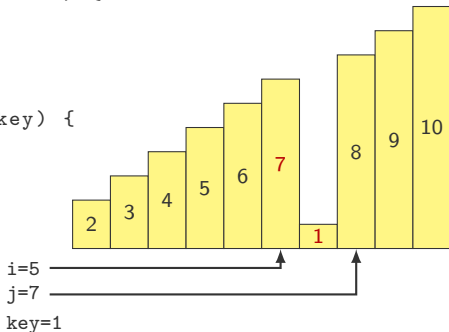
1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }
    
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

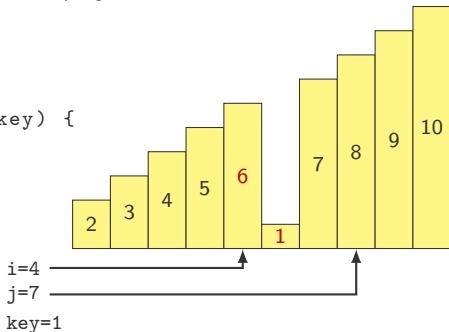
```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

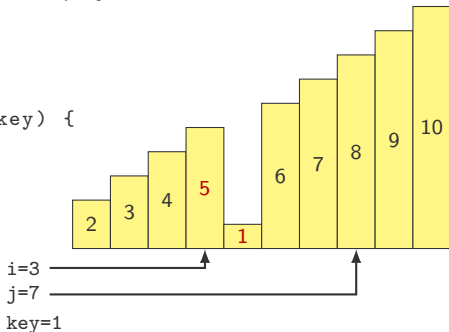
```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

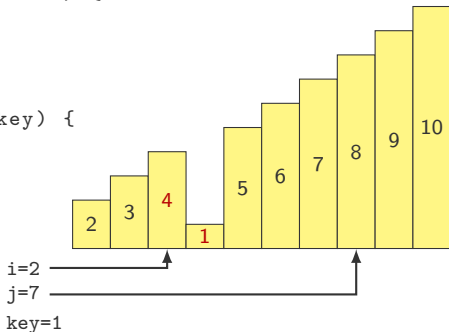
```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

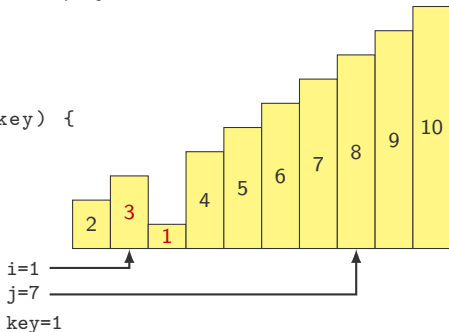
```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

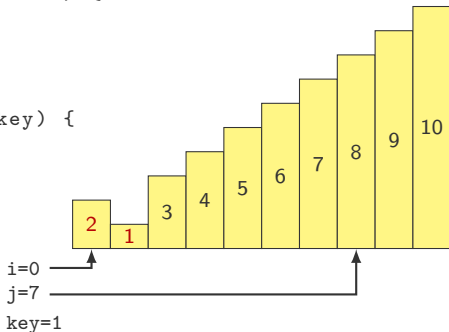
```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

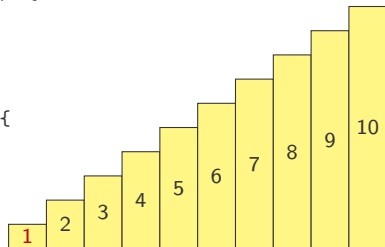
```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

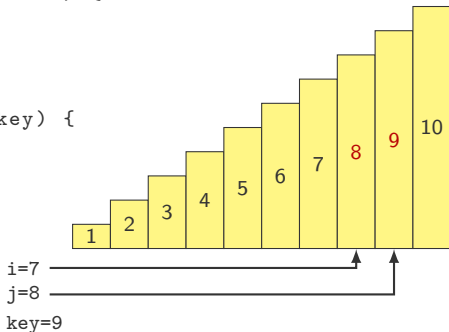
```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

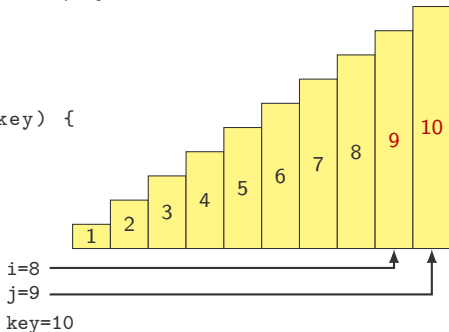
```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

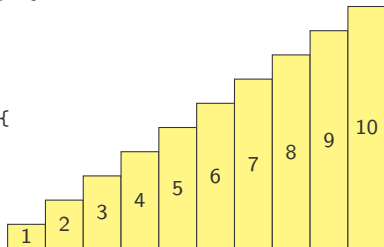
```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```



Ordenação por Inserção

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```



i
j
key

Ordenação por Inserção - Complexidade do algoritmo

```
1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }
```


Ordenação por Inserção - Complexidade do algoritmo

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```

- O consumo de tempo do `insertionSort` é proporcional ao número de execuções da comparação $A[i] > key$.

Ordenação por Inserção - Complexidade do algoritmo

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```

- O consumo de tempo do `insertionSort` é proporcional ao número de execuções da comparação $A[i] > key$.
- Para cada j , a variável i assume no máximo j valores: $j - 1, j - 2, \dots, 0$.

Ordenação por Inserção - Complexidade do algoritmo

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```

- O consumo de tempo do `insertionSort` é proporcional ao número de execuções da comparação $A[i] > key$.
- Para cada j , a variável i assume no máximo j valores: $j - 1, j - 2, \dots, 0$.
- Como $1 \leq j \leq n - 1$, o número de execuções da linha 6 é igual a $\sum_{j=1}^{n-1} j$ no pior caso.

Ordenação por Inserção - Complexidade do algoritmo

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```

- O consumo de tempo do `insertionSort` é proporcional ao número de execuções da comparação $A[i] > key$.
- Para cada j , a variável i assume no máximo j valores: $j-1, j-2, \dots, 0$.
- Como $1 \leq j \leq n-1$, o número de execuções da linha 6 é igual a $\sum_{j=1}^{n-1} j$ no pior caso.
- Essa soma é igual a $n(n-1)/2 = O(n^2)$.

Ordenação por Inserção - Complexidade do algoritmo

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```

Ordenação por Inserção - Complexidade do algoritmo

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```

No pior caso:

Ordenação por Inserção - Complexidade do algoritmo

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```

No pior caso:

- comparações (linha 6): $\approx n^2/2 = O(n^2)$

Ordenação por Inserção - Complexidade do algoritmo

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```

No pior caso:

- comparações (linha 6): $\approx n^2/2 = O(n^2)$
- atribuições (linha 7): $\approx n^2/2 = O(n^2)$



SelectionSort



Ordenação por Seleção

Ideia:

Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$

Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$

Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...

Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if(A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

Ordenação por Seleção

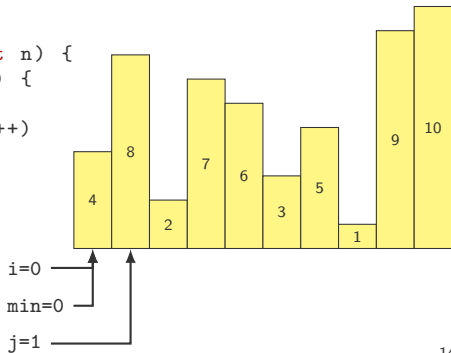
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```



Ordenação por Seleção

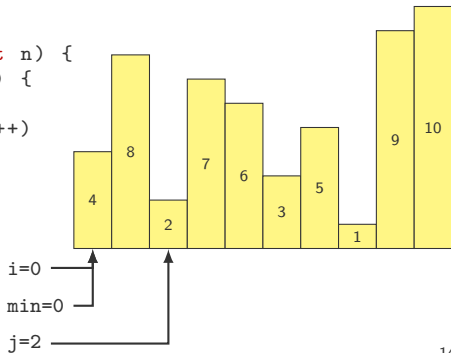
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```



Ordenação por Seleção

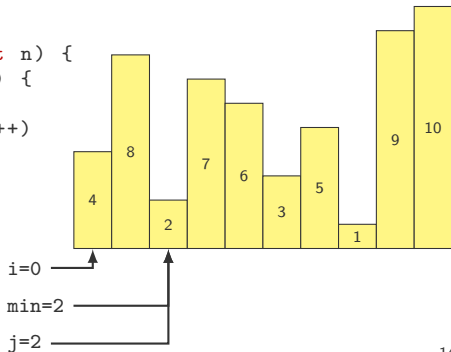
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```



Ordenação por Seleção

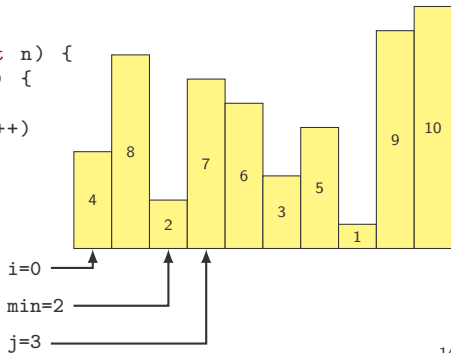
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```

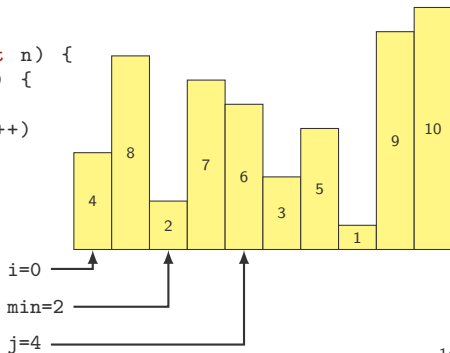


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```



Ordenação por Seleção

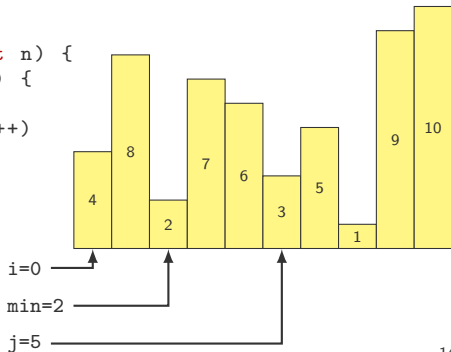
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```



Ordenação por Seleção

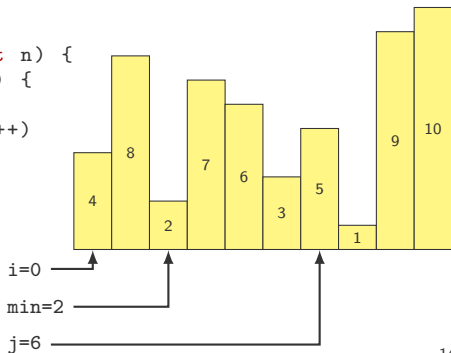
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```



Ordenação por Seleção

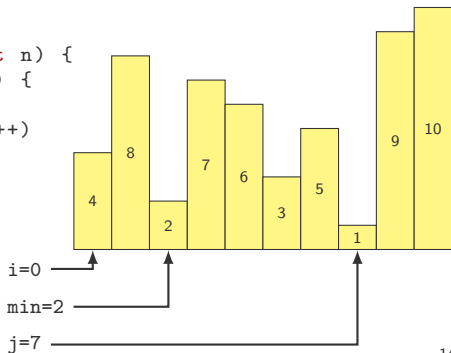
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```

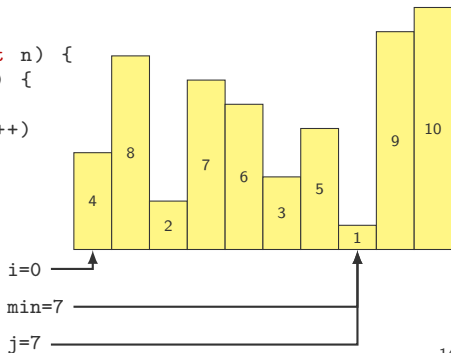


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```



Ordenação por Seleção

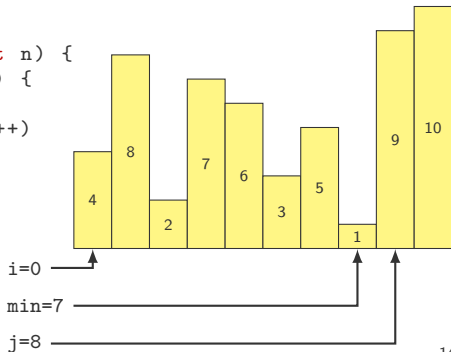
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```



Ordenação por Seleção

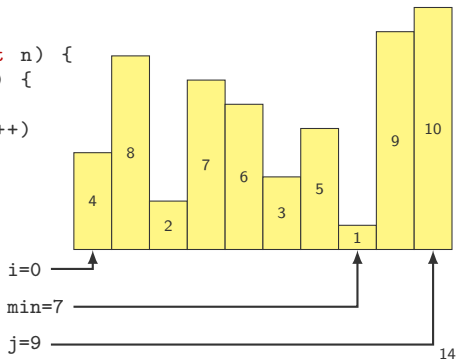
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```

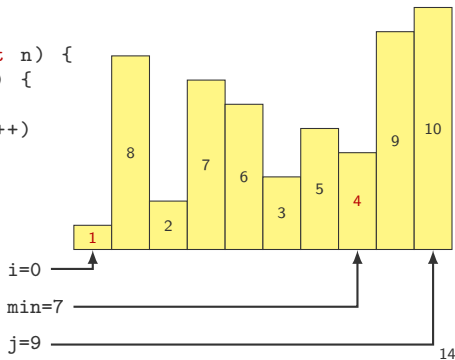


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

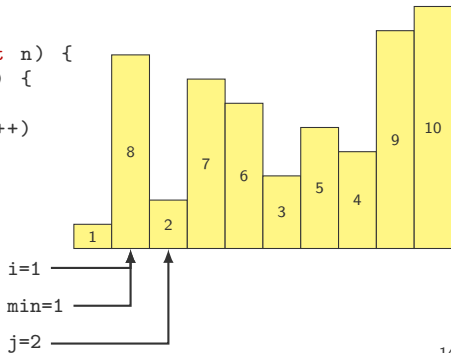


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

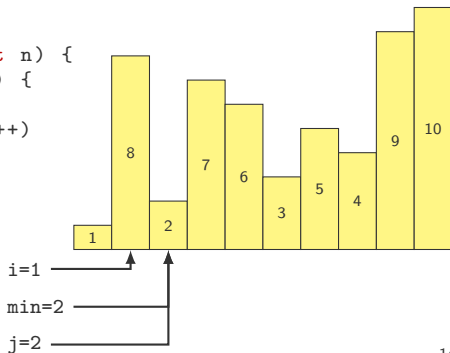


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

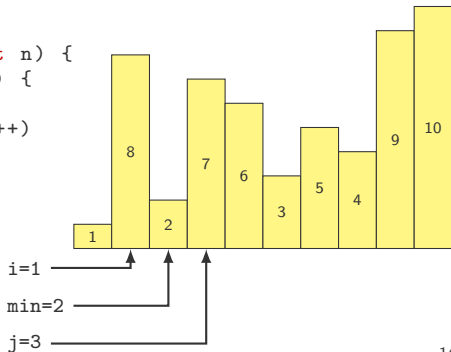


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```



Ordenação por Seleção

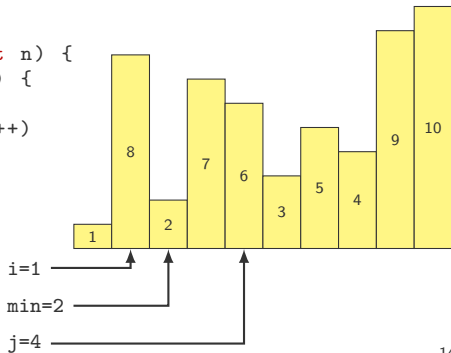
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```



Ordenação por Seleção

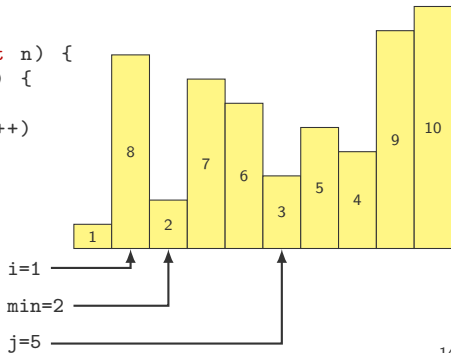
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```

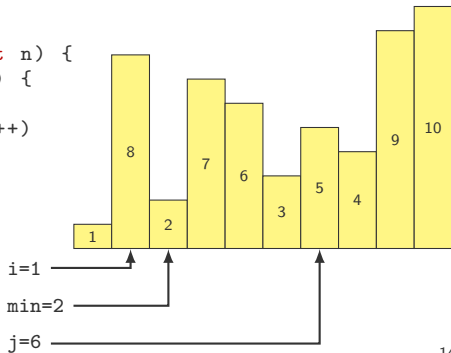


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```



Ordenação por Seleção

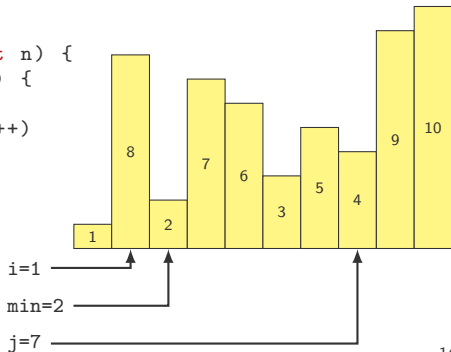
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```



Ordenação por Seleção

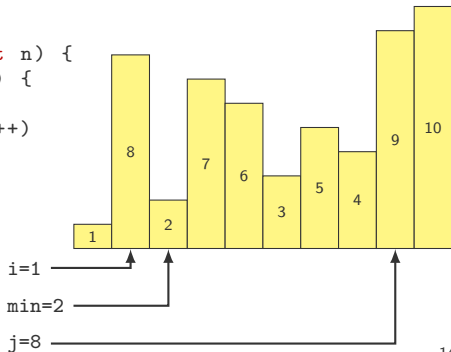
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```

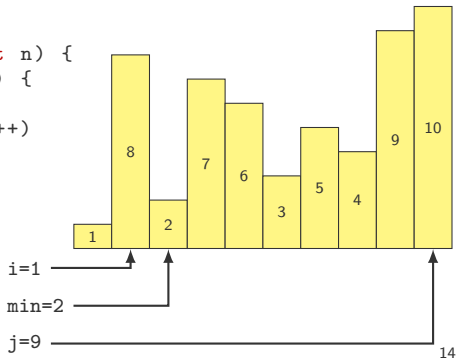


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

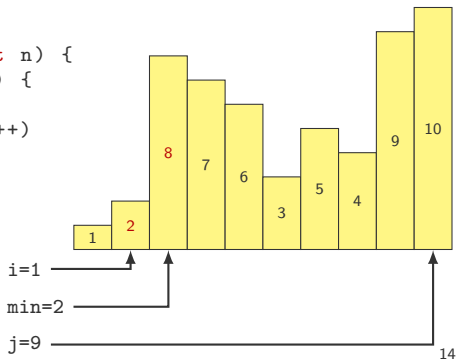


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

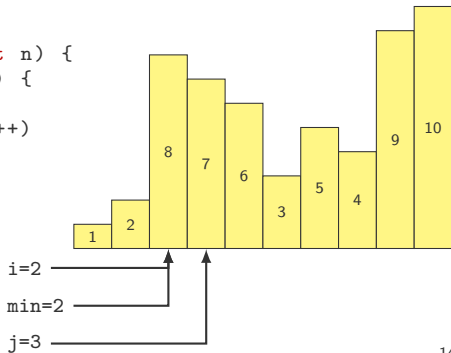


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```



Ordenação por Seleção

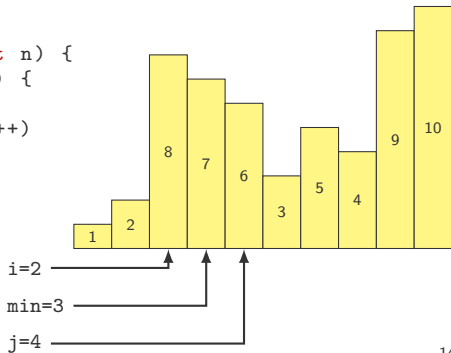
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```

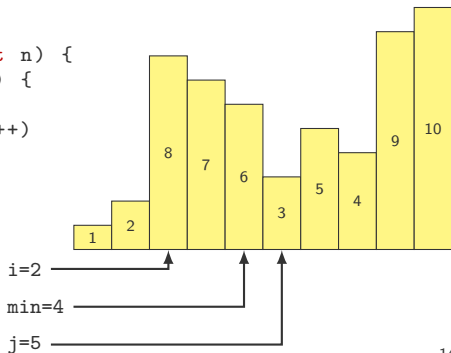


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```



Ordenação por Seleção

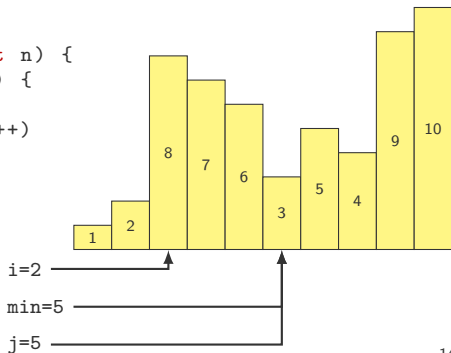
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```



Ordenação por Seleção

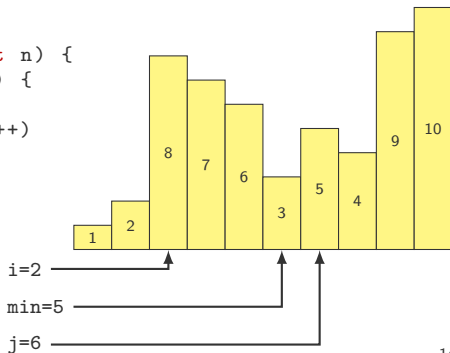
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```

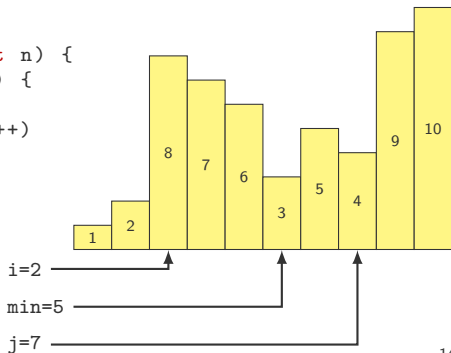


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```



Ordenação por Seleção

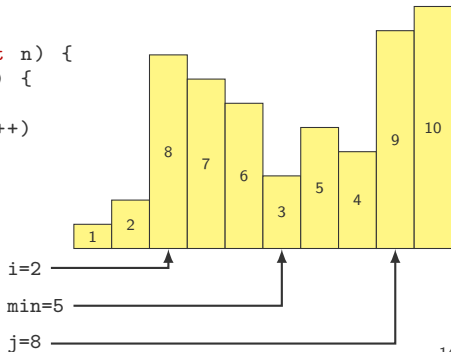
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```



Ordenação por Seleção

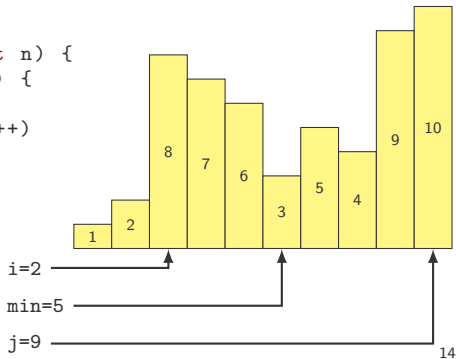
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```



Ordenação por Seleção

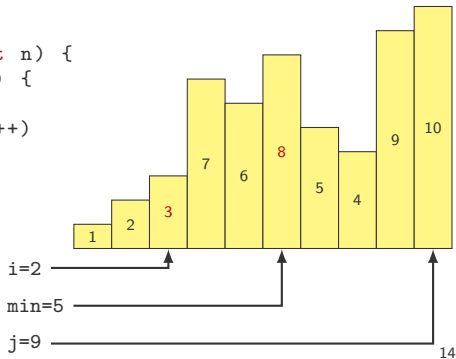
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```

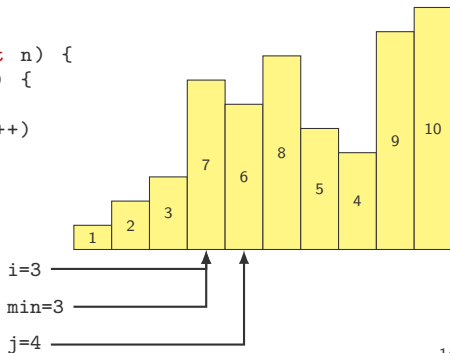


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

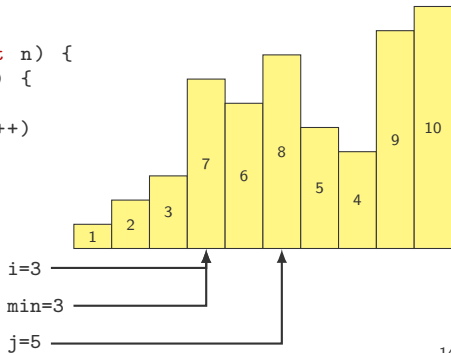


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

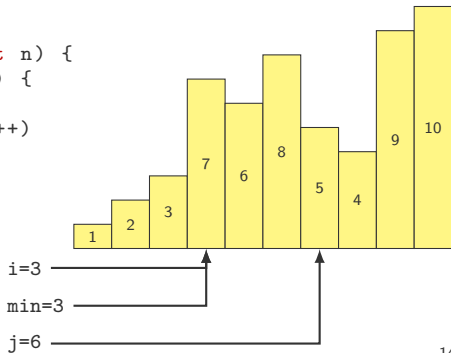


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

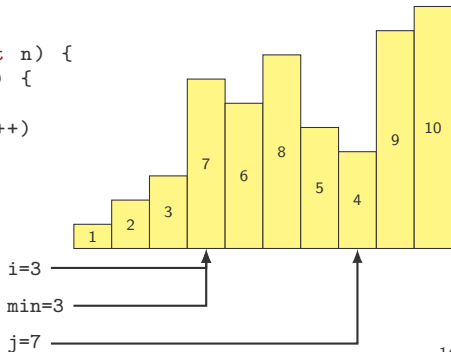


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

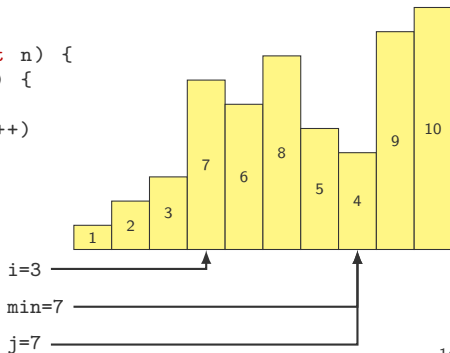


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

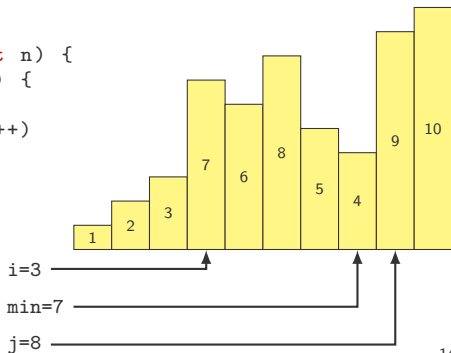


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

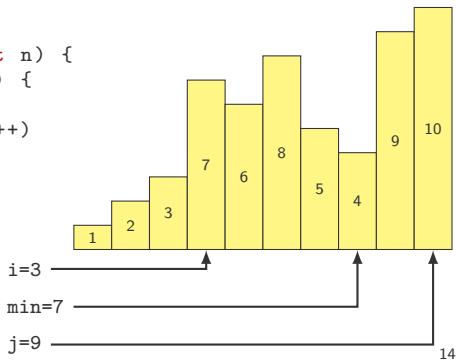


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

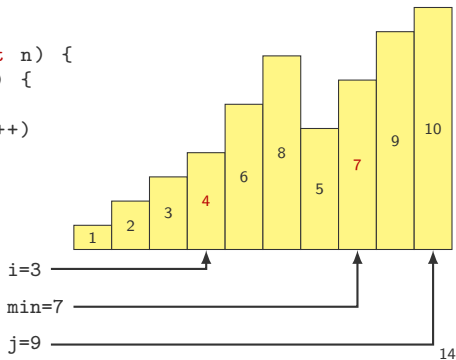


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```



Ordenação por Seleção

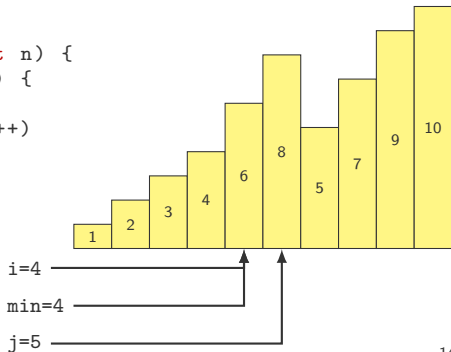
Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```

1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if (A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
9 }

```

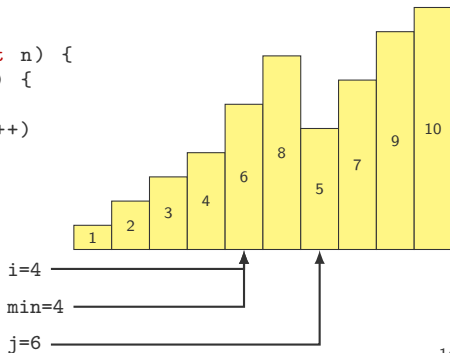


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

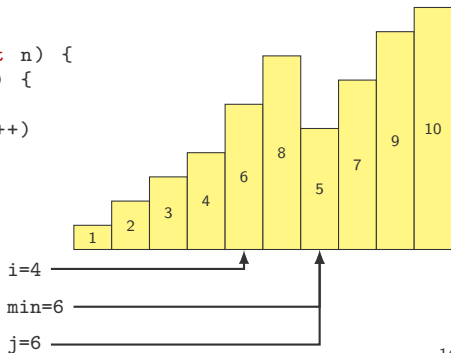


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

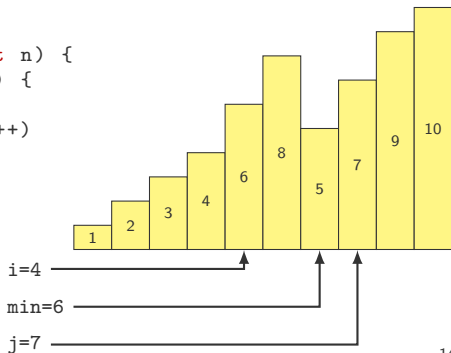


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

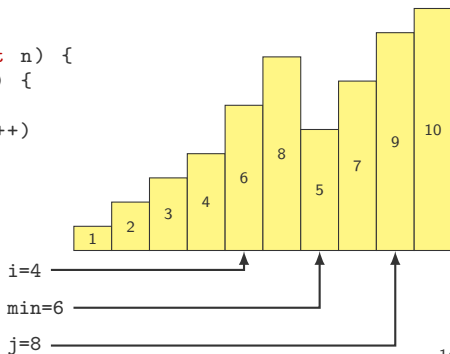


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

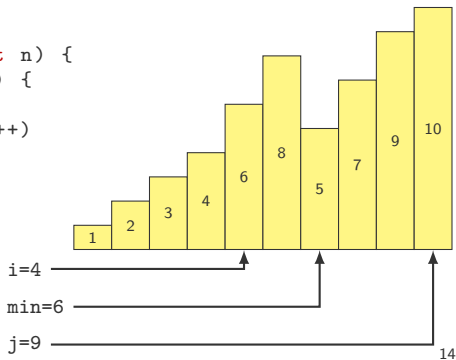


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

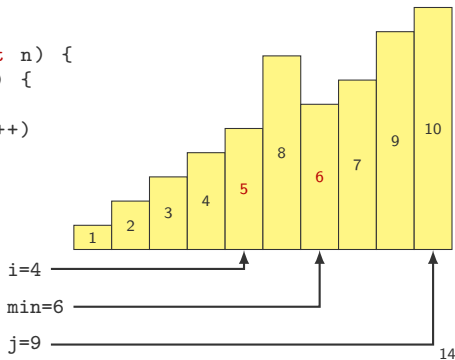


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

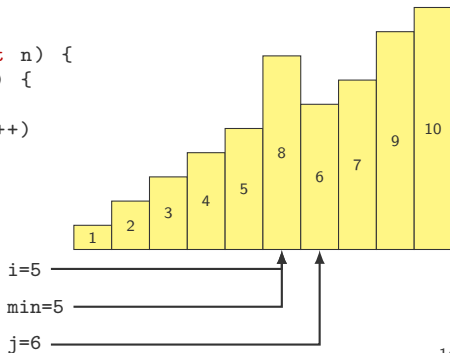


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

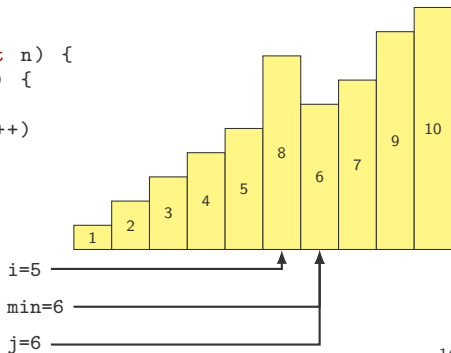


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

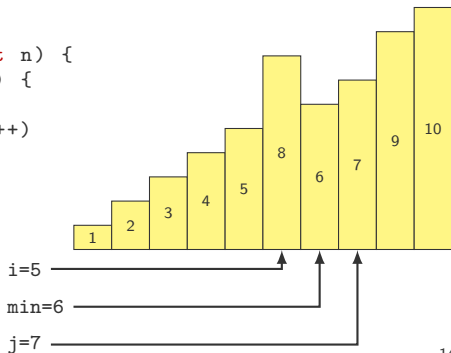


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

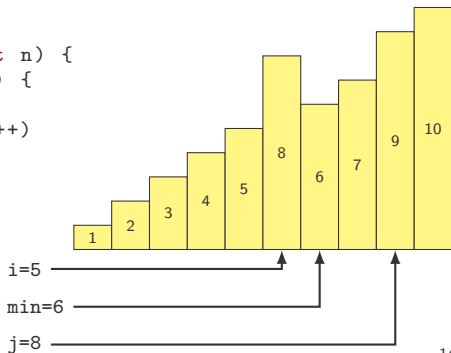


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

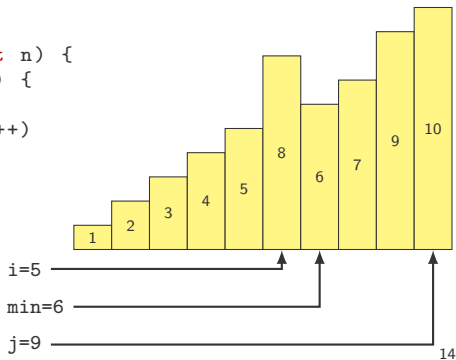


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

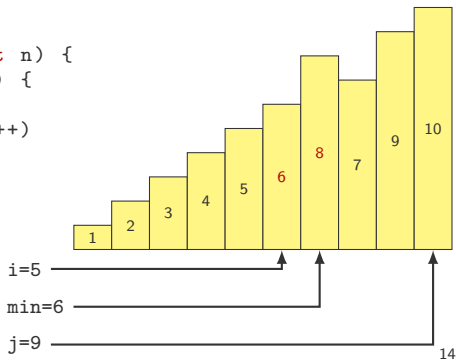


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

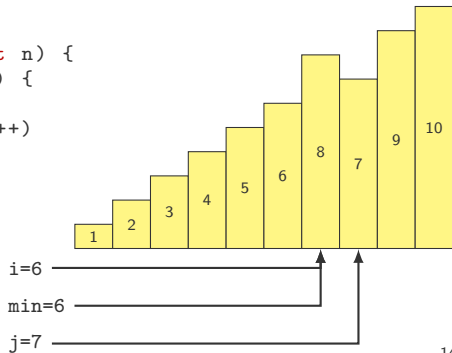


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

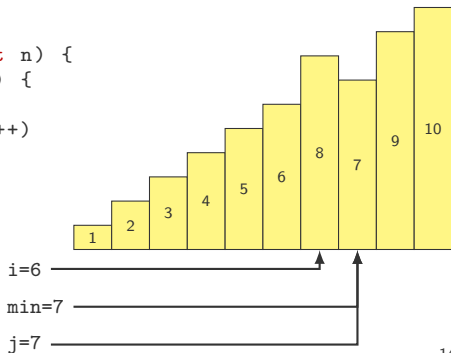


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

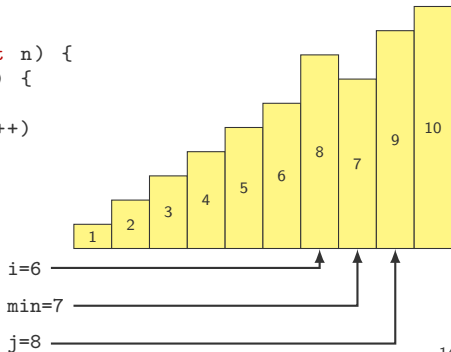


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

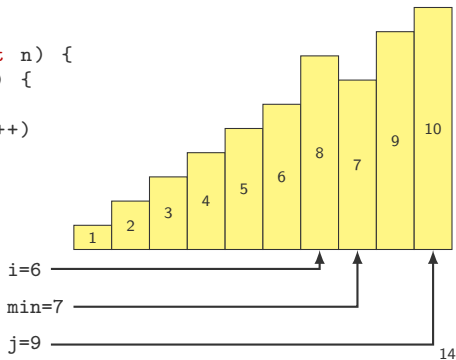


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

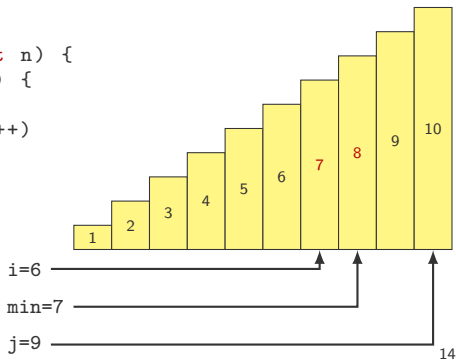


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

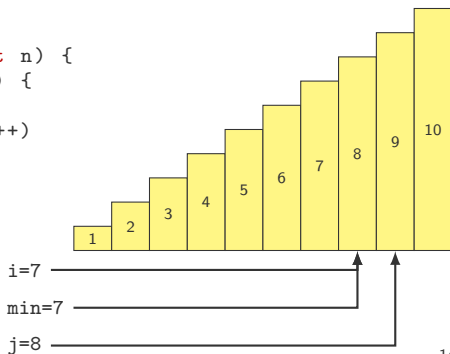


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

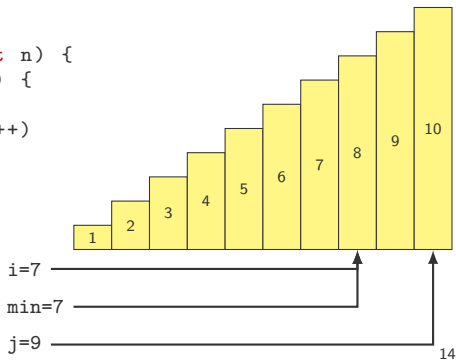


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

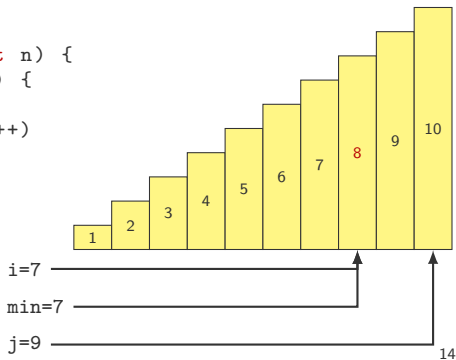


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

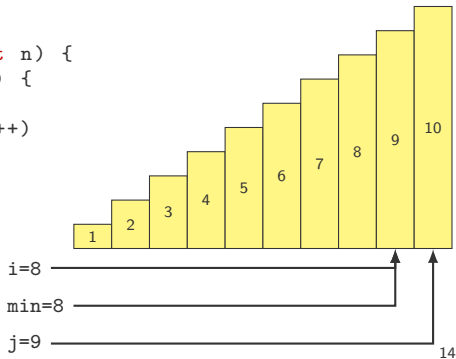


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

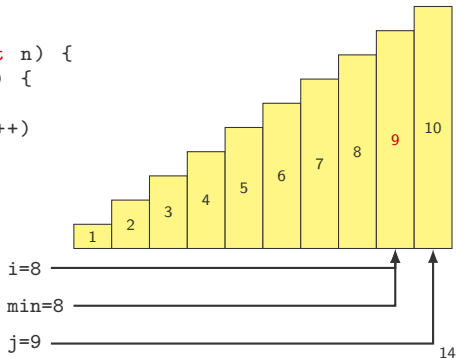


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

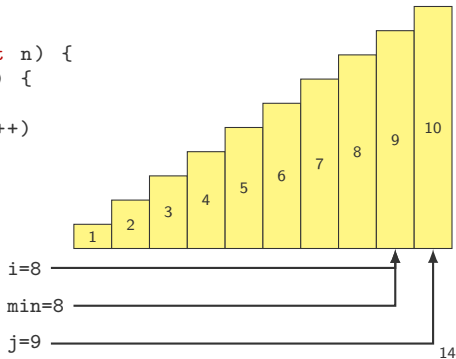


Ordenação por Seleção

Ideia:

- Trocar $v[0]$ com o mínimo de $v[0], v[1], \dots, v[n-1]$
- Trocar $v[1]$ com o mínimo de $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar $v[i]$ com o mínimo de $v[i], v[i+1], \dots, v[n-1]$

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if (A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```



Ordenação por Seleção – Complexidade do algoritmo

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if(A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i],A[min]);  
8     }  
9 }
```

Ordenação por Seleção – Complexidade do algoritmo

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if(A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

- número de comparações:

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

Ordenação por Seleção – Complexidade do algoritmo

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if(A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

- número de comparações:

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

- número de trocas: $n-1 = O(n)$

Algoritmos in Loco



Algoritmos in loco

- Um **algoritmo in loco** é um algoritmo que transforma a entrada sem usar estruturas de dados auxiliares.

Algoritmos in loco

- Um **algoritmo in loco** é um algoritmo que transforma a entrada sem usar estruturas de dados auxiliares.
- No entanto, uma pequena quantidade de espaço de armazenamento extra é permitida para variáveis auxiliares.

Algoritmos in loco

- Um **algoritmo in loco** é um algoritmo que transforma a entrada sem usar estruturas de dados auxiliares.
- No entanto, uma pequena quantidade de espaço de armazenamento extra é permitida para variáveis auxiliares.
- A entrada é geralmente sobrescrita pela saída conforme o algoritmo é executado. O algoritmo in loco atualiza a entrada apenas por meio da substituição ou troca de elementos.

Algoritmos in loco

- Um **algoritmo in loco** é um algoritmo que transforma a entrada sem usar estruturas de dados auxiliares.
- No entanto, uma pequena quantidade de espaço de armazenamento extra é permitida para variáveis auxiliares.
- A entrada é geralmente sobrescrita pela saída conforme o algoritmo é executado. O algoritmo in loco atualiza a entrada apenas por meio da substituição ou troca de elementos.
- **Pergunta:** BubbleSort, Insertion-Sort e Selection-Sort são algoritmos in loco?

Ordenação Estável



Ordenação estável

- Um algoritmo de ordenação é **estável** se não altera a posição relativa de elementos que têm um mesmo valor.

Ordenação estável

- Um algoritmo de ordenação é **estável** se não altera a posição relativa de elementos que têm um mesmo valor.
- Por exemplo, se o vetor tiver dois elementos de valor 13, um algoritmo de ordenação estável manterá o primeiro 13 antes do segundo.

Ordenação estável

- Um algoritmo de ordenação é **estável** se não altera a posição relativa de elementos que têm um mesmo valor.
- Por exemplo, se o vetor tiver dois elementos de valor 13, um algoritmo de ordenação estável manterá o primeiro 13 antes do segundo.
- **Exemplo:** Suponha que os elementos de um vetor são pares da forma (d, m) que representam datas de um certo ano: a primeira componente representa o dia e a segunda o mês.

Ordenação estável — Exemplo

- Suponha que o vetor está em ordem crescente das componentes d :
 $(1,12), (7,12), (16,3), (25,9), (30,3), (30,6), (31,3)$.

Ordenação estável — Exemplo

- Suponha que o vetor está em ordem crescente das componentes d :
 $(1,12), (7,12), (16,3), (25,9), (30,3), (30,6), (31,3)$.
- Agora ordene o vetor pelas componentes m . Se usarmos um algoritmo de ordenação estável, o resultado estará em ordem cronológica:
 $(16,3), (30,3), (31,3), (30,6), (25,9), (1,12), (7,12)$.

Ordenação estável — Exemplo

- Suponha que o vetor está em ordem crescente das componentes d :
 $(1,12), (7,12), (16,3), (25,9), (30,3), (30,6), (31,3)$.
- Agora ordene o vetor pelas componentes m . Se usarmos um algoritmo de ordenação estável, o resultado estará em ordem cronológica:
 $(16,3), (30,3), (31,3), (30,6), (25,9), (1,12), (7,12)$.
- Se o algoritmo de ordenação não for estável, o resultado pode não ficar em ordem cronológica:
 $(30,3), (16,3), (31,3), (30,6), (25,9), (7,12), (1,12)$.

Exercício – Ordenação estável

- BubbleSort é um algoritmo estável?

```
1 void bubblesort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++)  
3         for (int j = n-1; j > i; j--)  
4             if (A[j] < A[j-1])  
5                 std::swap(A[j], A[j-1]);  
6 }
```

Exercício – Ordenação estável

- Selection-Sort é um algoritmo estável?

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if(A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }
```

Exercício – Ordenação estável

- Selection-Sort é um algoritmo estável?

```
1 void selectionsort(int A[], int n) {
2     for (int i = 0; i < n-1; i++) {
3         int min = i;
4         for (int j = i+1; j < n; j++)
5             if(A[j] < A[min])
6                 min = j;
7         std::swap(A[i], A[min]);
8     }
```

- Insertion-Sort é um algoritmo estável?

```
1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
```


Invariantes e corretude de algoritmos



Invariantes

- O corpo de muitas funções contém um ou mais processos iterativos (tipicamente controlados por um laço **for** ou um **while**).

- O corpo de muitas funções contém um ou mais processos iterativos (tipicamente controlados por um laço **for** ou um **while**).
 - Por exemplo, a função `insertionsort` tem um laço `for` e um `while`.

- O corpo de muitas funções contém um ou mais processos iterativos (tipicamente controlados por um laço **for** ou um **while**).
 - Por exemplo, a função `insertionsort` tem um laço `for` e um `while`.
- Um **invariante** é uma relação entre os valores das variáveis que **vale no início de cada iteração do laço**.

- O corpo de muitas funções contém um ou mais processos iterativos (tipicamente controlados por um laço **for** ou um **while**).
 - Por exemplo, a função `insertionsort` tem um laço `for` e um `while`.
- Um **invariante** é uma relação entre os valores das variáveis que **vale no início de cada iteração do laço**.
- Os invariantes explicam o funcionamento do processo iterativo e permitem provar, por indução, que ele tem o efeito desejado.

Invariantes

- A fim de **provar um invariante de laço**, devemos mostrar que:

Invariantes

- A fim de **provar um invariante de laço**, devemos mostrar que:
 - Ele é válido antes do laço iniciar (**Inicialização**)

Invariantes

- A fim de **provar um invariante de laço**, devemos mostrar que:
 - Ele é válido antes do laço iniciar (**Inicialização**)
 - Supondo que ele é válido no início de uma iteração i qualquer, mostrar que ele continua válido no início da iteração $i + 1$ (**Manutenção**).

Invariantes

- A fim de **provar um invariante de laço**, devemos mostrar que:
 - Ele é válido antes do laço iniciar (**Inicialização**)
 - Supondo que ele é válido no início de uma iteração i qualquer, mostrar que ele continua válido no início da iteração $i + 1$ (**Manutenção**).
- Por fim, com estes dois fatos, deduzimos que:

- A fim de **provar um invariante de laço**, devemos mostrar que:
 - Ele é válido antes do laço iniciar (**Inicialização**)
 - Supondo que ele é válido no início de uma iteração i qualquer, mostrar que ele continua válido no início da iteração $i + 1$ (**Manutenção**).
- Por fim, com estes dois fatos, deduzimos que:
 - o invariante é válido no início da última iteração do laço (**Término**).

- A fim de **provar um invariante de laço**, devemos mostrar que:
 - Ele é válido antes do laço iniciar (**Inicialização**)
 - Supondo que ele é válido no início de uma iteração i qualquer, mostrar que ele continua válido no início da iteração $i + 1$ (**Manutenção**).
- Por fim, com estes dois fatos, deduzimos que:
 - o invariante é válido no início da última iteração do laço (**Término**).
- Com o invariante, mostramos que o processo iterativo faz o que se propôs a fazer.

Ordenação por Inserção - Corretude do algoritmo

```
1 void insertionsort(int A[], int n) {  
2     int i, j, key;  
3     for (j = 1; j < n; j++) {  
4         key = A[j];  
5         i = j-1;  
6         while (i >= 0 && A[i] > key) {  
7             A[i+1] = A[i];  
8             i--;  
9         }  
10        A[i+1] = key;  
11    }  
12 }
```

A fim de provar que o Insertion-Sort é **correto**, devemos provar que o seguinte invariante de laço é verdadeiro:

Ordenação por Inserção - Corretude do algoritmo

```
1 void insertionsort(int A[], int n) {
2     int i, j, key;
3     for (j = 1; j < n; j++) {
4         key = A[j];
5         i = j-1;
6         while (i >= 0 && A[i] > key) {
7             A[i+1] = A[i];
8             i--;
9         }
10        A[i+1] = key;
11    }
12 }
```

A fim de provar que o Insertion-Sort é **correto**, devemos provar que o seguinte invariante de laço é verdadeiro:

- No início de cada iteração do laço **for** das linhas 3-11, o subvetor $A[0 \dots j-1]$ consiste nos elementos que estavam originalmente em $A[0 \dots j-1]$, porém em sequência ordenada.

Exercícios



Exercício – Corretude do BubbleSort

Prove que o algoritmo **Bubblesort** está correto mostrando que os dois invariantes de laço a seguir são verdadeiros:

- (a) No início de cada iteração do laço **for** das linhas 2-4, o subvetor $A[j \dots n - 1]$ consiste em uma permutação dos valores que estavam originalmente em $A[j \dots n - 1]$ antes do laço iniciar.

Além disso, $A[j] = \min\{A[k] : j \leq k \leq n - 1\}$, ou seja, o elemento na posição $A[j]$ é o menor dentre todos os elementos nesse subvetor.

Exercício – Corretude do BubbleSort

Prove que o algoritmo **Bubblesort** está correto mostrando que os dois invariantes de laço a seguir são verdadeiros:

- (a) No início de cada iteração do laço **for** das linhas 2-4, o subvetor $A[j \dots n - 1]$ consiste em uma permutação dos valores que estavam originalmente em $A[j \dots n - 1]$ antes do laço iniciar.
- Além disso, $A[j] = \min\{A[k] : j \leq k \leq n - 1\}$, ou seja, o elemento na posição $A[j]$ é o menor dentre todos os elementos nesse subvetor.
- (b) No início de cada iteração do laço **for** das linhas 1-4, o subarray $A[0 \dots i - 1]$ consiste nos i menores valores originalmente em $A[0 \dots n - 1]$, ordenados em ordem crescente, e $A[i \dots n - 1]$ consiste nos $n - i$ valores restantes originalmente em $A[0 \dots n - 1]$.

Exercício – Corretude do SelectionSort

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if(A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

A fim de provar que o **selectionsort** é **correto**, devemos provar que o seguinte invariante de laço é verdadeiro:

Exercício – Corretude do SelectionSort

```
1 void selectionsort(int A[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         int min = i;  
4         for (int j = i+1; j < n; j++)  
5             if(A[j] < A[min])  
6                 min = j;  
7         std::swap(A[i], A[min]);  
8     }  
9 }
```

A fim de provar que o **selectionsort** é **correto**, devemos provar que o seguinte invariante de laço é verdadeiro:

- No início de cada iteração do laço **for** das linhas 2-8, o subvetor $A[0 \dots i - 1]$ contém os i menores elementos do vetor original e o subvetor $A[i \dots n - 1]$ contém os $n - i$ maiores elementos do vetor original. Além disso, o subvetor $A[0 \dots i - 1]$ é crescente.

FIM

