Estruturas de Dados e Básicas I - DIM0119

Selan R. dos Santos

DIMAp – Departamento de Informática e Matemática Aplicada
Sala 231, ramal 231, selan@dimap.ufrn.br
UFRN

2018.1

Algoritmos de Ordenação — Conteúdo

Motivação

- Atualmente podemos afirmar que estamos vivendo a era da informação, na qual é importante saber guardar e recuperar informações de uma maneira que faça sentido.

Motivação

- Atualmente podemos afirmar que estamos vivendo a era da informação, na qual é importante saber guardar e recuperar informações de uma maneira que faça sentido.
- Há algum tempo dizia-se que metade do tempo de processamento de computadores comerciais era dedicado a ordenação de dados—hoje isso não é mais verdade em função da alta capacidade de processamento dos computadores mordernos e seus algoritmos.

Motivação

- Atualmente podemos afirmar que estamos vivendo a era da informação, na qual é importante saber guardar e recuperar informações de uma maneira que faça sentido.
- Há algum tempo dizia-se que metade do tempo de processamento de computadores comerciais era dedicado a ordenação de dados—hoje isso não é mais verdade em função da alta capacidade de processamento dos computadores mordernos e seus algoritmos.
- Contudo, ainda assim a maioria das aplicações precisam, de alguma maneira, apresentar seus dados aos usuário seguindo algum tipo de ordenação.

Motivação

- Atualmente podemos afirmar que estamos vivendo a era da informação, na qual é importante saber guardar e recuperar informações de uma maneira que faça sentido.
- Há algum tempo dizia-se que metade do tempo de processamento de computadores comerciais era dedicado a ordenação de dados—hoje isso não é mais verdade em função da alta capacidade de processamento dos computadores mordernos e seus algoritmos.
- Contudo, ainda assim a maioria das aplicações precisam, de alguma maneira, apresentar seus dados aos usuário seguindo algum tipo de ordenação.
- É por estes motivos que foram desenvolvidos diversos algoritmos de ordenação, seguindo várias estratégias diferentes para ordenar elementos.

Objetivos

- Vamos estudar apenas alguns algoritmos de ordenação, escolhidos por:

Objetivos

- ▻ Vamos estudar apenas alguns algoritmos de ordenação, escolhidos por:
  - ⋆ Serem bons algoritmos—cada um pode ser a melhor escolha sob determinadas circusntâncias;

Objetivos

- Vamos estudar apenas alguns algoritmos de ordenação, escolhidos por:
  - Serem bons algoritmos—cada um pode ser a melhor escolha sob determinadas circusntâncias;
  - Ilustrarem muitas das variedades de algoritmos existentes; e

Objetivos

- Vamos estudar apenas alguns algoritmos de ordenação, escolhidos por:
  - Serem bons algoritmos—cada um pode ser a melhor escolha sob determinadas circusntâncias;
  - Ilustrarem muitas das variedades de algoritmos existentes; e
  - Serem relativamente simples de entender e fácil de escrever.

Objetivos

- Vamos estudar apenas alguns algoritmos de ordenação, escolhidos por:
  - Serem bons algoritmos—cada um pode ser a melhor escolha sob determinadas circusntâncias;
  - Ilustrarem muitas das variedades de algoritmos existentes; e
  - Serem relativamente simples de entender e fácil de escrever.
- Mas primeiramente precisamos definir alguns conceitos centrais para a tarefa de ordenação: **ordem total** e **ordem total estrita**.

Definição Ordem Total e Ordem Total Estrita

- Seja X um conjunto.

Definição Ordem Total e Ordem Total Estrita

- Seja X um conjunto.
- Uma ordem total em X é uma *relação binária* (denotada aqui por $\leqslant$) em X que satisfaz as seguintes propriedades para **todo** $a, b, c \in X$:

Definição Ordem Total e Ordem Total Estrita

- Seja X um conjunto.
- Uma ordem total em X é uma *relação binária* (denotada aqui por $\leqslant$) em X que satisfaz as seguintes propriedades para **todo** $a, b, c \in X$:
  - **Anti-simetria:** se $a \leqslant b$ e $b \leqslant a$ então $a = b$
    Anti-simetria elimina casos onde tanto a precede b quanto b precede a. Ou seja, dois elementos distintos não podem ser relacionados em ambas as direções

Definição Ordem Total e Ordem Total Estrita

- Seja X um conjunto.
- Uma ordem total em X é uma *relação binária* (denotada aqui por $\leqslant$) em X que satisfaz as seguintes propriedades para **todo** $a, b, c \in X$:
  - **Anti-simetria:** se $a \leqslant b$ e $b \leqslant a$ então $a = b$
    Anti-simetria elimina casos onde tanto a precede b quanto b precede a. Ou seja, dois elementos distintos não podem ser relacionados em ambas as direções
  - **Transitividade:** se $a \leqslant b$ e $b \leqslant c$ então $a \leqslant c$
    O relacionamento pode ser propagado.

Definição Ordem Total e Ordem Total Estrita

- Seja X um conjunto.
- Uma ordem total em X é uma *relação binária* (denotada aqui por $\leqslant$) em X que satisfaz as seguintes propriedades para **todo** $a, b, c \in X$:
  - **Anti-simetria:** se $a \leqslant b$ e $b \leqslant a$ então $a = b$
    Anti-simetria elimina casos onde tanto a precede b quanto b precede a. Ou seja, dois elementos distintos não podem ser relacionados em ambas as direções
  - **Transitividade:** se $a \leqslant b$ e $b \leqslant c$ então $a \leqslant c$
    O relacionamento pode ser propagado.
  - **Totalidade:** $a \leqslant b$ ou $b \leqslant a$
    Qualquer par de elementos no conjunto da relação são comparáveis

Definição Ordem Total e Ordem Total Estrita

- Qual é a diferença entre ordem total e ordem parcial?

Definição Ordem Total e Ordem Total Estrita

- ▻ Qual é a diferença entre ordem total e ordem parcial?
- ▻ *R: a propriedade de totalidade.*

Definição Ordem Total e Ordem Total Estrita

- Qual é a diferença entre ordem total e ordem parcial?
- *R: a propriedade de totalidade.*
- Totalidade implica em reflexividade ($a \leqslant a$, para todo $a \in X$).

Definição Ordem Total e Ordem Total Estrita

- Qual é a diferença entre ordem total e ordem parcial?
- *R: a propriedade de totalidade.*
- Totalidade implica em reflexividade ($a \leqslant a$, para todo $a \in X$).
- Em uma ordem parcial nem todo par de elementos são relacionáveis (comparáveis).

Definição Ordem Total e Ordem Total Estrita

- Qual é a diferença entre ordem total e ordem parcial?
- *R: a propriedade de totalidade.*
- Totalidade implica em reflexividade ($a \leqslant a$, para todo $a \in X$).
- Em uma ordem parcial nem todo par de elementos são relacionáveis (comparáveis).
- Já em um conjunto sobre o qual a relação de ordem total está definida, quaisquer dois elementos podem ser mutualmente comparados com respeito à relação $\leqslant$.

Definição Ordem Total e Ordem Total Estrita

- Para cada ordem total $\leqslant$, existe uma relação assimétrica associada a ela, denotada aqui por $<$, chamada de ordem total estrita.

Definição Ordem Total e Ordem Total Estrita

- Para cada ordem total $\leqslant$, existe uma relação assimétrica associada a ela, denotada aqui por $<$, chamada de ordem total estrita.
- Uma ordem total estrita satisfaz a seguinte propriedade:
  $a < b$ se e somente se $a \leqslant b$ e $a \neq b$, $\forall a, b \in X$

Definição Ordem Total e Ordem Total Estrita

- Para cada ordem total $\leqslant$, existe uma relação assimétrica associada a ela, denotada aqui por $<$, chamada de ordem total estrita.
- Uma ordem total estrita satisfaz a seguinte propriedade:
  $a < b$ se e somente se $a \leqslant b$ e $a \neq b$, $\forall a, b \in X$
- A relação $<$ é transitiva.

- Para cada ordem total $\leqslant$, existe uma relação assimétrica associada a ela, denotada aqui por $<$, chamada de ordem total estrita.
- Uma ordem total estrita satisfaz a seguinte propriedade:
  $a < b$ se e somente se $a \leqslant b$ e $a \neq b$, $\forall a, b \in X$
- A relação $<$ é transitiva.
- Além disso, para quaisquer $a, b \in X$, apenas uma das três possibilidades ocorre:

- Para cada ordem total $\leqslant$, existe uma relação assimétrica associada a ela, denotada aqui por $<$, chamada de ordem total estrita.
- Uma ordem total estrita satisfaz a seguinte propriedade: $a < b$ se e somente se $a \leqslant b$ e $a \neq b$, $\forall a, b \in X$
- A relação $<$ é transitiva.
- Além disso, para quaisquer $a, b \in X$, apenas uma das três possibilidades ocorre:
  - $a < b$;

Definição Ordem Total e Ordem Total Estrita

- Para cada ordem total $\leqslant$, existe uma relação assimétrica associada a ela, denotada aqui por $<$, chamada de ordem total estrita.
- Uma ordem total estrita satisfaz a seguinte propriedade:
  $a < b$ se e somente se $a \leqslant b$ e $a \neq b$, $\forall a, b \in X$
- A relação $<$ é transitiva.
- Além disso, para quaisquer $a, b \in X$, apenas uma das três possibilidades ocorre:
  - $a < b$;
  - $b < a$; ou

Definição Ordem Total e Ordem Total Estrita

- Para cada ordem total $\leqslant$, existe uma relação assimétrica associada a ela, denotada aqui por $<$, chamada de ordem total estrita.
- Uma ordem total estrita satisfaz a seguinte propriedade:
  $a < b$ se e somente se $a \leqslant b$ e $a \neq b$, $\forall a, b \in X$
- A relação $<$ é transitiva.
- Além disso, para quaisquer $a, b \in X$, apenas uma das três possibilidades ocorre:
  - $a < b$;
  - $b < a$; ou
  - $a = b$.

- Se X admite uma ordem total então X é um conjunto totalmente ordenado.

Definição Ordem Total e Ordem Total Estrita

- Se X admite uma ordem total então X é um conjunto totalmente ordenado.
- O seguintes conjuntos admitem uma ordem total:

Definição Ordem Total e Ordem Total Estrita

- Se X admite uma ordem total então X é um conjunto totalmente ordenado.
- O seguintes conjuntos admitem uma ordem total:
  - O conjunto das letras do alfabeto ordenadas pela ordem lexicográfica do dicionário; i.e. $A < B < C < D < \cdots < Z$.

- Se X admite uma ordem total então X é um conjunto totalmente ordenado.
- O seguintes conjuntos admitem uma ordem total:
  - O conjunto das letras do alfabeto ordenadas pela ordem lexicográfica do dicionário; i.e. $A < B < C < D < \cdots < Z$.
  - O conjunto dos números reais onde $<$ é a relação *"é menor"*.

Definição Ordem Total e Ordem Total Estrita

- Se X admite uma ordem total então X é um conjunto totalmente ordenado.
- O seguintes conjuntos admitem uma ordem total:
  - O conjunto das letras do alfabeto ordenadas pela ordem lexicográfica do dicionário; i.e. $A < B < C < D < \cdots < Z$.
  - O conjunto dos números reais onde $<$ é a relação *"é menor"*.
  - Qualquer subconjunto de um conjunto totalmente ordenado com a mesma restrição de ordem do conjunto original.

Definição Ordem Total e Ordem Total Estrita

- Se X admite uma ordem total então X é um conjunto totalmente ordenado.
- O seguintes conjuntos admitem uma ordem total:
  - O conjunto das letras do alfabeto ordenadas pela ordem lexicográfica do dicionário; i.e. $A < B < C < D < \cdots < Z$.
  - O conjunto dos números reais onde $<$ é a relação *"é menor"*.
  - Qualquer subconjunto de um conjunto totalmente ordenado com a mesma restrição de ordem do conjunto original.
  - Os conjuntos dos números naturais, inteiros e racionais com a relação $<$ ("é menor") do conjunto dos números reais.

Entrada:

Uma sequência,

$$\langle a_1, \ldots, a_n \rangle,$$

de n elementos, com $n \in \mathbb{Z}^+$, tal que os elementos da sequência pertencem a um conjunto totalmente ordenável por '$\leqslant$'.

O Problema de Ordenação

Entrada:

Uma sequência,

$$\langle a_1, \ldots, a_n \rangle,$$

de n elementos, com $n \in \mathbb{Z}^+$, tal que os elementos da sequência pertencem a um conjunto totalmente ordenável por '$\leqslant$'.

Saída:

Uma permutação,

$$\langle a_{\pi 1}, \ldots, a_{\pi n} \rangle,$$

da sequência de entrada tal que

$$a_{\pi 1} \leqslant a_{\pi 2} \leqslant \cdots \leqslant a_{\pi n}.$$

O Problema de Ordenação

Importante:

Para quaisquer dois elementos, a_i e a_j, da sequência S, temos que apenas uma das três seguintes afirmações deve ser verdadeira:

$$a_i < a_j, \quad a_j < a_i \quad \text{ou} \quad a_i = a_j,$$

onde $<$ é qualquer ordem total estrita associada com a ordem total $\leqslant$.

Algoritmos de Ordenação

- Vamos estudar algoritmos para resolver o problema de ordenação.

Algoritmos de Ordenação

- Vamos estudar algoritmos para resolver o problema de ordenação.
- Cada um deles recebe como entrada um inteiro não-negativo, n, e um vetor, A, com os n elementos da sequência, S, de entrada.

- Vamos estudar algoritmos para resolver o problema de ordenação.
- Cada um deles recebe como entrada um inteiro não-negativo, n, e um vetor, A, com os n elementos da sequência, S, de entrada.
- Nós supomos existir uma função, denominada `compara`, que recebe dois elementos, a e b, de S como entrada e retorna

$$\begin{array}{rl} -1 & \text{se } a < b\,, \\ 0 & \text{se } a = b\,, \\ 1 & \text{se } b < a\,. \end{array}$$

- Vamos estudar algoritmos para resolver o problema de ordenação.
- Cada um deles recebe como entrada um inteiro não-negativo, n, e um vetor, A, com os n elementos da sequência, S, de entrada.
- Nós supomos existir uma função, denominada `compara`, que recebe dois elementos, a e b, de S como entrada e retorna

$$\begin{array}{rl} -1 & \text{se } a < b\,, \\ 0 & \text{se } a = b\,, \\ 1 & \text{se } b < a\,. \end{array}$$

- Uma versão mais simples para `compara` recebe dois elementos, a e b, de S e retorna **vedadeiro** se $a < b$. Com ela é possível realizar os três testes acima.

- ▷ Na definição de `compara`, o símbolo '$<$' se refere à relação de ordem total estrita que o algoritmo de ordenação usará para ordenar os elementos de S.

- Na definição de `compara`, o símbolo '$<$' se refere à relação de ordem total estrita que o algoritmo de ordenação usará para ordenar os elementos de S.
  - Qualquer ordem total estrita pode ser usada.

- Na definição de `compara`, o símbolo '$<$' se refere à relação de ordem total estrita que o algoritmo de ordenação usará para ordenar os elementos de S.
  - Qualquer ordem total estrita pode ser usada.
  - Basta codificá-la em `compara`.

- Na definição de `compara`, o símbolo '$<$' se refere à relação de ordem total estrita que o algoritmo de ordenação usará para ordenar os elementos de S.
  - ⋆ Qualquer ordem total estrita pode ser usada.
  - ⋆ Basta codificá-la em `compara`.
- A lógica dos algoritmos de ordenação que veremos não depende do tipo dos elementos de S nem da relação de ordem total estrita adotada.

Ordenação por Comparação

- Assim, algoritmos de ordenação do tipo **comparação** são aqueles que aplicam uma função abstratata `compara()` sobre seus elementos para determinar a ordem final da lista.

Ordenação por Comparação

- Assim, algoritmos de ordenação do tipo **comparação** são aqueles que aplicam uma função abstratata `compara()` sobre seus elementos para determinar a ordem final da lista.
- Em contraste, temos os algoritmos que **não usam comparação**, mas sim algum tipo de análise de chave que determina a posição final do elemento (sem precisar "ver" os demais).

- Assim, algoritmos de ordenação do tipo **comparação** são aqueles que aplicam uma função abstratata `compara()` sobre seus elementos para determinar a ordem final da lista.
- Em contraste, temos os algoritmos que **não usam comparação**, mas sim algum tipo de análise de chave que determina a posição final do elemento (sem precisar "ver" os demais).
- Três algoritmos que se enquadram na categoria comparação:

- ▷ Assim, algoritmos de ordenação do tipo **comparação** são aqueles que aplicam uma função abstratata `compara()` sobre seus elementos para determinar a ordem final da lista.
- ▷ Em contraste, temos os algoritmos que **não usam comparação**, mas sim algum tipo de análise de chave que determina a posição final do elemento (sem precisar "ver" os demais).
- ▷ Três algoritmos que se enquadram na categoria comparação:
  - ⋆ Inserção ordenada

- ▷ Assim, algoritmos de ordenação do tipo **comparação** são aqueles que aplicam uma função abstratata `compara()` sobre seus elementos para determinar a ordem final da lista.
- ▷ Em contraste, temos os algoritmos que **não usam comparação**, mas sim algum tipo de análise de chave que determina a posição final do elemento (sem precisar "ver" os demais).
- ▷ Três algoritmos que se enquadram na categoria comparação:
  - ⋆ Inserção ordenada
  - ⋆ *Insertion sort*

- Assim, algoritmos de ordenação do tipo **comparação** são aqueles que aplicam uma função abstratata `compara()` sobre seus elementos para determinar a ordem final da lista.
- Em contraste, temos os algoritmos que **não usam comparação**, mas sim algum tipo de análise de chave que determina a posição final do elemento (sem precisar "ver" os demais).
- Três algoritmos que se enquadram na categoria comparação:
  - Inserção ordenada
  - *Insertion sort*
  - *Selection sort*

- Ligeiramente diferente dos demais algoritmos: a estrutura é ordenada no decorrer da inserção de seus elementos.

- Ligeiramente diferente dos demais algoritmos: a estrutura é ordenada no decorrer da inserção de seus elementos.
  - Antes da inserção de um elemento: a estrutura está ordenada.

- Ligeiramente diferente dos demais algoritmos: a estrutura é ordenada no decorrer da inserção de seus elementos.
    - Antes da inserção de um elemento: a estrutura está ordenada.
    - Depois da inserção de um elemento: a estrutura está ordenada.

- Ligeiramente diferente dos demais algoritmos: a estrutura é ordenada no decorrer da inserção de seus elementos.
  - Antes da inserção de um elemento: a estrutura está ordenada.
  - Depois da inserção de um elemento: a estrutura está ordenada.
  - A estrutra ordenada, portanto, é o invariante do laço.

Com memória sequencial (a.k.a. arranjo)

- É necessário deslocar elementos para dar espaço para a inserção.

Com memória sequencial (a.k.a. arranjo)

- É necessário deslocar elementos para dar espaço para a inserção.
- Procedimento:

Com memória sequencial (a.k.a. arranjo)

- É necessário deslocar elementos para dar espaço para a inserção.
- Procedimento:
  1. Encontrar o local de inserção por:

Com memória sequencial (a.k.a. arranjo)

- É necessário deslocar elementos para dar espaço para a inserção.
- Procedimento:
  1. Encontrar o local de inserção por:
     - Método #1: busca binária.

Com memória sequencial (a.k.a. arranjo)

- É necessário deslocar elementos para dar espaço para a inserção.
- Procedimento:
  1. Encontrar o local de inserção por:
     - Método #1: busca binária.
     - Método #2: busca sequencial a partir do fim do vetor.

Com memória sequencial (a.k.a. arranjo)

- É necessário deslocar elementos para dar espaço para a inserção.
- Procedimento:
  1. Encontrar o local de inserção por:
     - Método #1: busca binária.
     - Método #2: busca sequencial a partir do fim do vetor.
  2. Deslocar os elementos seguintes ao local de inserção.

Com memória sequencial (a.k.a. arranjo)

- É necessário deslocar elementos para dar espaço para a inserção.
- Procedimento:
  1. Encontrar o local de inserção por:
     - Método #1: busca binária.
     - Método #2: busca sequencial a partir do fim do vetor.
  2. Deslocar os elementos seguintes ao local de inserção.
  3. Inserir novo elemento no local de inserção.

Com memória sequencial (a.k.a. arranjo)

- É necessário deslocar elementos para dar espaço para a inserção.
- Procedimento:
  1. Encontrar o local de inserção por:
     - Método #1: busca binária.
     - Método #2: busca sequencial a partir do fim do vetor.
  2. Deslocar os elementos seguintes ao local de inserção.
  3. Inserir novo elemento no local de inserção.

Com memória encadeada (a.k.a. lista encadeada)

Com memória sequencial (a.k.a. arranjo)

- É necessário deslocar elementos para dar espaço para a inserção.
- Procedimento:
  1. Encontrar o local de inserção por:
     - Método #1: busca binária.
     - Método #2: busca sequencial a partir do fim do vetor.
  2. Deslocar os elementos seguintes ao local de inserção.
  3. Inserir novo elemento no local de inserção.

Com memória encadeada (a.k.a. lista encadeada)

- Não é ncessário deslocar elementos seguintes ao local de inserção.

Com memória sequencial (a.k.a. arranjo)

- É necessário deslocar elementos para dar espaço para a inserção.
- Procedimento:
  1. Encontrar o local de inserção por:
     - Método #1: busca binária.
     - Método #2: busca sequencial a partir do fim do vetor.
  2. Deslocar os elementos seguintes ao local de inserção.
  3. Inserir novo elemento no local de inserção.

Com memória encadeada (a.k.a. lista encadeada)

- Não é ncessário deslocar elementos seguintes ao local de inserção.
- Baseado na modificação dos apontadores que cercam o local de inserção.

Novo elemento

6

3	5	7	8	9		

Lista ordenada

Novo elemento

6

$6 < 9?$

3	**5**	**7**	**8**	**9**		

Lista ordenada

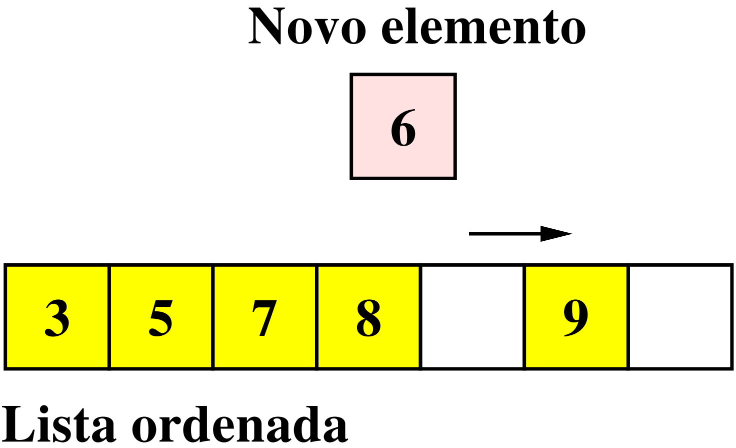
Novo elemento

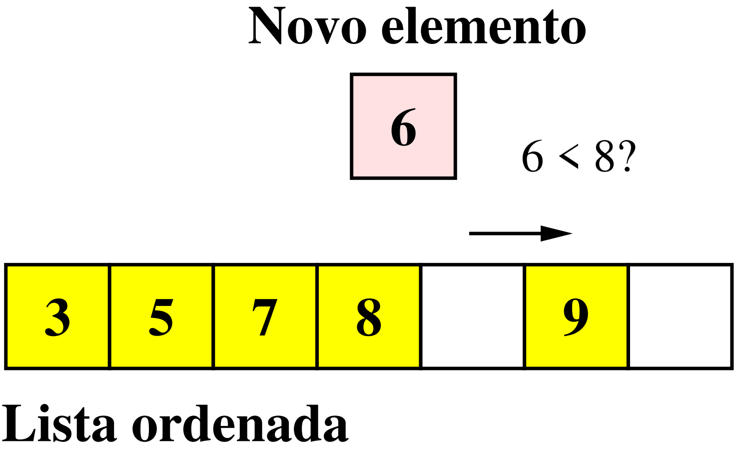
6

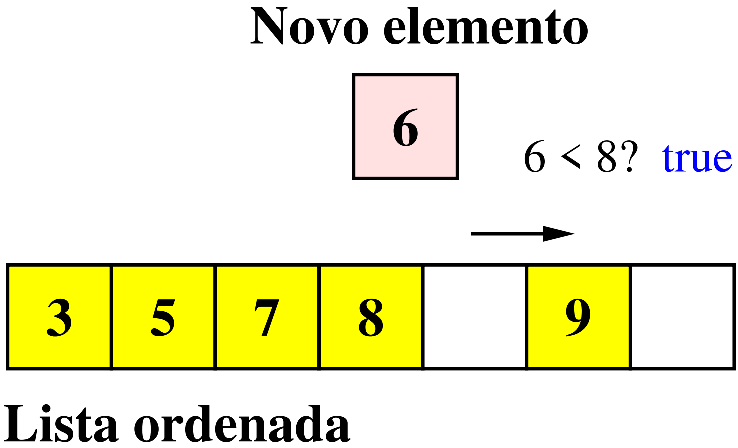
$6 < 9$? true

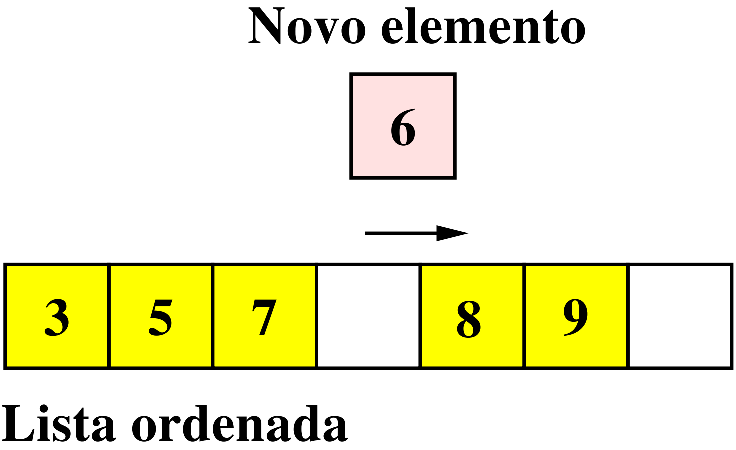
3	**5**	**7**	**8**	**9**		

Lista ordenada

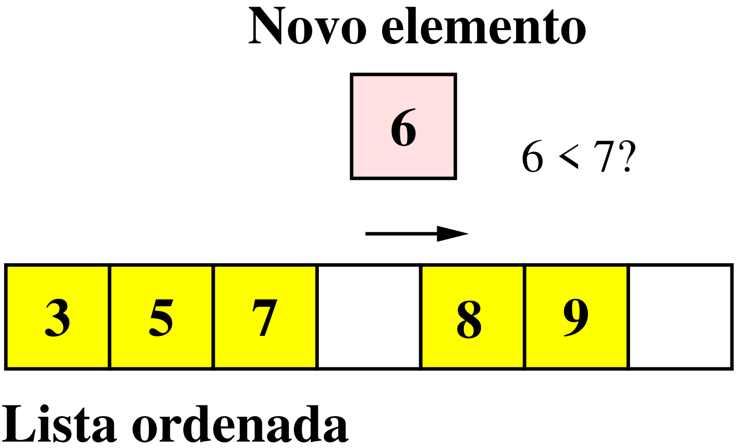

Novo elemento
6
3
5
7
8
9
Lista ordenada

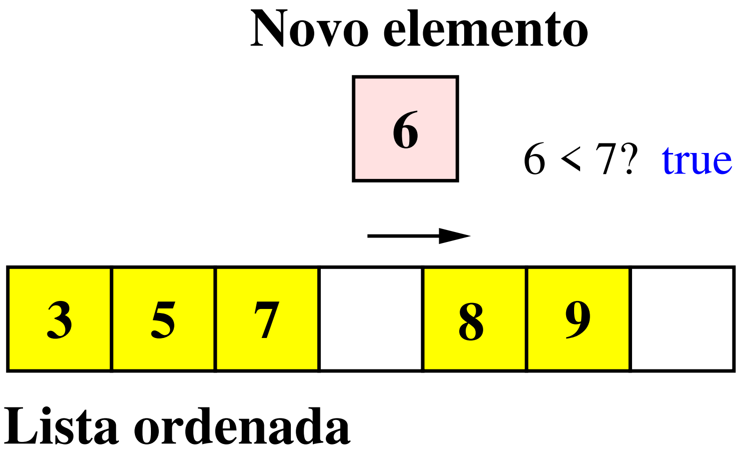

Novo elemento
6
6 < 8?
3
5
7
8
9
Lista ordenada

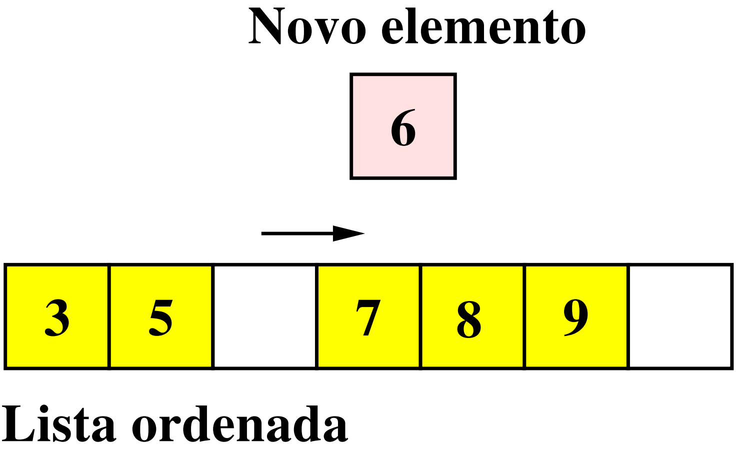

Novo elemento
6
6 < 8? true
3
5
7
8
9
Lista ordenada



Novo elemento
6
3
5
7
8
9
Lista ordenada

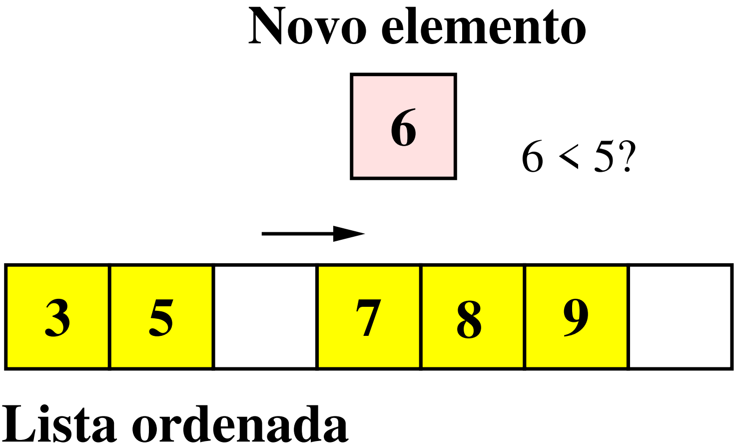

Novo elemento
6
6 < 7?
3
5
7
8
9
Lista ordenada

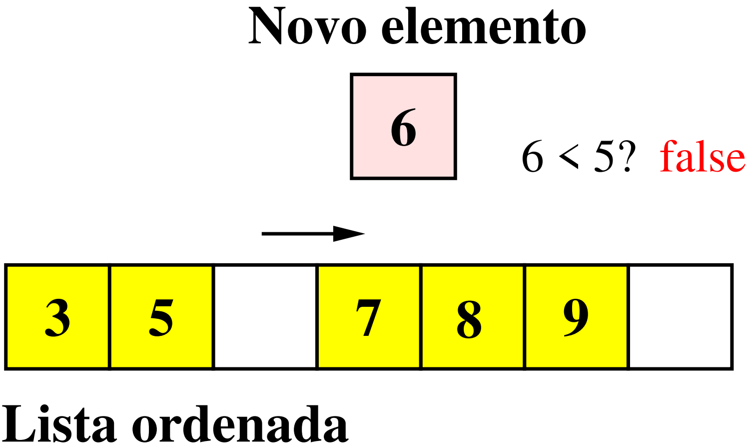

Novo elemento
6
6 < 7? true
3
5
7
8
9
Lista ordenada

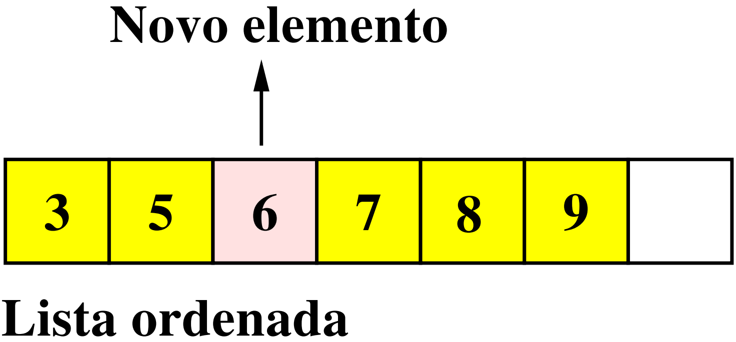

Novo elemento
6
3
5
7
8
9
Lista ordenada

Ordenação por Inserção

(1) Algoritmo Inserção Ordenada — exemplo




Novo elemento
6
6 < 5? false
3
5
7
8
9
Lista ordenada


Novo elemento
3
5
6
7
8
9
Lista ordenada

O que aconteceria neste caso?

Novo elemento

6b

3	5	6a	7	8	9	

Lista ordenada

O que aconteceria neste caso?

3	**5**	**6a**	**6b**	**7**	**8**	**9**

ou

3	**5**	**6b**	**6a**	**7**	**8**	**9**

Lista ordenada

```cpp
// A     -> The array.
// size  -> The current number of elements in the array.
// N     -> The array's max capacity.
// value -> The element we want to insert in A.
void insert ( int A[], int & size, int N, int value ) {
    if ( size == N )    return; // Array is already full!

    auto i( size-1 ); // Start at the last element.
    while ( i >= 0 && compare( value, A[i] ) < 0 )
    {
        // Move (copy) element forward,
        A[i+1] = A[i]; // ... opening a new "hole" in array.
        i--; // Move towards the array's begining.
    }
    // Store new element at the right position, ...
    A[ i+1 ] = value;  // preserving the sorting order.

    size++;
}
```

Ordenação por Inserção

(1) Algoritmo Inserção Ordenada — corretude

```
// {P: ‘A is sorted’ & ‘size <= N’ & ‘value MIGHT NOT BE in A’ }
void insert ( int A[], int & size, int N, int value ) {
    if ( size == N )    return;

    auto i( size-1 );
    // { IL: value < A[k], for all k>i }
    // [ 1 1 2 2 3 ][ ], i = 4, value = 1, A[k>4]=empty; IL is true.
    while ( i >= 0 && compare( value, A[i] ) < 0 )
    {
        A[i+1] = A[i];
        i--;
        // [ 1 1 2 2 _ ][3], i = 3, 1 < [3]; IL is true.
    }
    // [ 1 1 _ ][2 2 3], i = 1, 1 < [2 2 3]; IL is true.
    A[ i+1 ] = value;
    // [ 1 1 1 ][2 2 3]
    size++;
}
// { Q: ‘A is sorted’ & ‘size <= N’ & ‘value IS in A’ }
```

Ordenação por Inserção

(1) Algoritmo Inserção Ordenada — versão ranges

```cpp
int * insert ( int *first, int *last, const int &new_item_,
               bool ( *compare )(int, int) )
{
    auto it( last-1 ); // Start at the last valid element.
    while ( i >= first and compare( value , *i ) )
    {
        // Move (copy) element forward,
        *(it+1) = std::move( *it ); //... opening "hole" in range.
        it--; // Move towards the begining of the array.
    }
    // Store new element at the right position, ...
    *(it+1) = value;  // preserving the sorting order.

    return last+1;
}
```

- Realiza a ordenação a partir de uma *coleção de elementos* que podem ou não estar ordenados.

- Realiza a ordenação a partir de uma *coleção de elementos* que podem ou não estar ordenados.
- Inicialmente separamos a coleção a ser ordenada em duas regiões, uma já ordenada, inicialmente vazia mas que cresce, e outra (supostamente) não-ordenada, inicialmente do tamanho n mas que vai sendo reduzida.

- Realiza a ordenação a partir de uma *coleção de elementos* que podem ou não estar ordenados.
- Inicialmente separamos a coleção a ser ordenada em duas regiões, uma já ordenada, inicialmente vazia mas que cresce, e outra (supostamente) não-ordenada, inicialmente do tamanho n mas que vai sendo reduzida.
- Algoritmo em alto nível:

- Realiza a ordenação a partir de uma *coleção de elementos* que podem ou não estar ordenados.
- Inicialmente separamos a coleção a ser ordenada em duas regiões, uma já ordenada, inicialmente vazia mas que cresce, e outra (supostamente) não-ordenada, inicialmente do tamanho n mas que vai sendo reduzida.
- Algoritmo em alto nível:
  1. Para todo item da parte não-ordenada faça:

- Realiza a ordenação a partir de uma *coleção de elementos* que podem ou não estar ordenados.
- Inicialmente separamos a coleção a ser ordenada em duas regiões, uma já ordenada, inicialmente vazia mas que cresce, e outra (supostamente) não-ordenada, inicialmente do tamanho n mas que vai sendo reduzida.
- Algoritmo em alto nível:
  1. Para todo item da parte não-ordenada faça:
     - Procurar sua (nova) posição na parte já ordenada;

Ordenação por Inserção

(2) Algoritmo **Insertion Sort**

- Realiza a ordenação a partir de uma *coleção de elementos* que podem ou não estar ordenados.
- Inicialmente separamos a coleção a ser ordenada em duas regiões, uma já ordenada, inicialmente vazia mas que cresce, e outra (supostamente) não-ordenada, inicialmente do tamanho n mas que vai sendo reduzida.
- Algoritmo em alto nível:
  1. Para todo item da parte não-ordenada faça:
     - Procurar sua (nova) posição na parte já ordenada;
     - Inserir o item na posição apropriada na parte ordenada.

- Realiza a ordenação a partir de uma *coleção de elementos* que podem ou não estar ordenados.
- Inicialmente separamos a coleção a ser ordenada em duas regiões, uma já ordenada, inicialmente vazia mas que cresce, e outra (supostamente) não-ordenada, inicialmente do tamanho n mas que vai sendo reduzida.
- Algoritmo em alto nível:
  1. Para todo item da parte não-ordenada faça:
     - Procurar sua (nova) posição na parte já ordenada;
     - Inserir o item na posição apropriada na parte ordenada.
- *Lembra do problema filtragem em vetor?*

- Realiza a ordenação a partir de uma *coleção de elementos* que podem ou não estar ordenados.
- Inicialmente separamos a coleção a ser ordenada em duas regiões, uma já ordenada, inicialmente vazia mas que cresce, e outra (supostamente) não-ordenada, inicialmente do tamanho n mas que vai sendo reduzida.
- Algoritmo em alto nível:
  1. Para todo item da parte não-ordenada faça:
     - Procurar sua (nova) posição na parte já ordenada;
     - Inserir o item na posição apropriada na parte ordenada.
- *Lembra do problema filtragem em vetor?*
- Note que para vetores, a procura também envolve deslocar elementos para abrir espaço para inserção.

▻ Considere Vetor A: $\boxed{5}\boxed{2}\boxed{4}\boxed{6}\boxed{1}\boxed{3}$

- Considere Vetor A:

5	2	4	6	1	3

- A função `compara()` implementa $<$ (é menor) para números reais.

Iteração #0 ⇒ A: $\boxed{5}\boxed{2}\boxed{4}\boxed{6}\boxed{1}\boxed{3}$

Iteração #1 $\Rightarrow A$: $\boxed{5}$ $\boxed{2}\boxed{4}\boxed{6}\boxed{1}\boxed{3}$

Iteração #2 ⇒ A: | 2 | 5 | | 4 | 6 | 1 | 3 |

Iteração #3 ⇒ A: $\boxed{2}\boxed{4}\boxed{5}\ \boxed{6}\boxed{1}\boxed{3}$

Iteração #4 ⇒ A: | 2 | 4 | 5 | 6 | | 1 | 3 |

Iteração #5 ⇒ A: | 1 | 2 | 4 | 5 | 6 | | 3 |

Iteração #6 ⇒ A:

1	2	3	4	5	6

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — pseudocódigo

Insertion Sort

```
procedimento insertionSort(A: arranjo de ref inteiro)
    var n: inteiro ← tam A                                      # recuperar tamanho vetor
    var key, holePos, i: inteiro                                # variáveis auxiliares
    para i ← 1 até n − 1 faça                                   # percorrer parte não-ordenada
        key ← A[i]                      # 1o item da parte não-ordenada (a ser inserido)
        holePos ← i − 1                   # começar a procurar do final da parte ordenada
        # percorrer (reverso) parte ordenada até início ou achar posição
        enquanto holePos ⩾ 0 e compara(key, A[holePos]) < 0 faça
            A[holePos + 1] ← A[holePos]                         # elementos p/ frente
            holePos ← holePos − 1                               # continuar procura
        A[holePos + 1] ← key                                    # inserir item na posição criada
```

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — análise

```
4:  para i ← 1 até n − 1 faça                          # percorrer parte não-ordenada
5:      key ← A[i]                     # 1o item da parte não-ordenada (a ser inserido)
6:      holePos ← i − 1                  # começar a procurar do final da parte ordenada
7:      enquanto holePos ⩾ 0 e compara(key, A[holePos]) < 0 faça
8:          A[holePos + 1] ← A[holePos]                   # deslocar elementos p/ frente
9:          holePos ← holePos − 1                                  # continuar procura
10:     A[holePos + 1] ← key                         # inserir item na posição criada
```

- O laço externo (linha 4) é executado $n-1$ vezes.

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — análise

```
4:  para i ← 1 até n − 1 faça                                  # percorrer parte não-ordenada
5:      key ← A[i]                                 # 1o item da parte não-ordenada (a ser inserido)
6:      holePos ← i − 1                              # começar a procurar do final da parte ordenada
7:      enquanto holePos ⩾ 0 e compara(key, A[holePos]) < 0 faça
8:          A[holePos + 1] ← A[holePos]                            # deslocar elementos p/ frente
9:          holePos ← holePos − 1                                          # continuar procura
10:     A[holePos + 1] ← key                                  # inserir item na posição criada
```

- O laço externo (linha 4) é executado $n - 1$ vezes.
- E o laço interno?

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — análise

```
4:  para i ← 1 até n − 1 faça                                 #percorrer parte não-ordenada
5:      key ← A[i]                             #1o item da parte não-ordenada (a ser inserido)
6:      holePos ← i − 1                          #começar a procurar do final da parte ordenada
7:      enquanto holePos ⩾ 0 e compara(key, A[holePos]) < 0 faça
8:          A[holePos + 1] ← A[holePos]                         #deslocar elementos p/ frente
9:          holePos ← holePos − 1                                          #continuar procura
10:     A[holePos + 1] ← key                                  #inserir item na posição criada
```

- O laço externo (linha 4) é executado $n - 1$ vezes.
- E o laço interno?
  - ⋆ **Melhor caso**: laço interno nunca é executado.

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — análise

```
4:  para i ← 1 até n − 1 faça                                  #percorrer parte não-ordenada
5:      key ← A[i]                          #1o item da parte não-ordenada (a ser inserido)
6:      holePos ← i − 1                        #começar a procurar do final da parte ordenada
7:      enquanto holePos ⩾ 0 e compara(key, A[holePos]) < 0 faça
8:          A[holePos + 1] ← A[holePos]                            #deslocar elementos p/ frente
9:          holePos ← holePos − 1                                           #continuar procura
10:     A[holePos + 1] ← key                                  #inserir item na posição criada
```

- O laço externo (linha 4) é executado $n - 1$ vezes.
- E o laço interno?
  - **Melhor caso**: laço interno nunca é executado.
    - Em que situação isso ocorre?

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — análise

```
4:  para i ← 1 até n − 1 faça                                   # percorrer parte não-ordenada
5:      key ← A[i]                                  # 1o item da parte não-ordenada (a ser inserido)
6:      holePos ← i − 1                              # começar a procurar do final da parte ordenada
7:      enquanto holePos ⩾ 0 e compara(key, A[holePos]) < 0 faça
8:          A[holePos + 1] ← A[holePos]                            # deslocar elementos p/ frente
9:          holePos ← holePos − 1                                             # continuar procura
10:     A[holePos + 1] ← key                                      # inserir item na posição criada
```

- O laço externo (linha 4) é executado $n - 1$ vezes.
- E o laço interno?
  - **Melhor caso**: laço interno nunca é executado.
    - Em que situação isso ocorre? *Arranjo em ordem crescente.*

4: **para** $i \leftarrow 1$ **até** $n-1$ **faça** `#percorrer parte não-ordenada`
5: $\quad key \leftarrow A[i]$ `#1o item da parte não-ordenada (a ser inserido)`
6: $\quad holePos \leftarrow i - 1$ `#começar a procurar do final da parte ordenada`
7: $\quad$ **enquanto** $holePos \geqslant 0$ **e** `compara`$(key,\ A[holePos]) < 0$ **faça**
8: $\quad\quad A[holePos+1] \leftarrow A[holePos]$ `#deslocar elementos p/ frente`
9: $\quad\quad holePos \leftarrow holePos - 1$ `#continuar procura`
10: $\quad A[holePos+1] \leftarrow key$ `#inserir item na posição criada`

- O laço externo (linha 4) é executado $n-1$ vezes.
- E o laço interno?
  - **Melhor caso**: laço interno nunca é executado.
    - Em que situação isso ocorre? *Arranjo em ordem crescente.*
  - **Pior caso**: laço interno é executado i vezes para cada iteração do laço externo.

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — análise

```
4:  para i ← 1 até n − 1 faça                                        # percorrer parte não-ordenada
5:      key ← A[i]                                   # 1o item da parte não-ordenada (a ser inserido)
6:      holePos ← i − 1                                # começar a procurar do final da parte ordenada
7:      enquanto holePos ⩾ 0 e compara(key, A[holePos]) < 0 faça
8:          A[holePos + 1] ← A[holePos]                                    # deslocar elementos p/ frente
9:          holePos ← holePos − 1                                                    # continuar procura
10:     A[holePos + 1] ← key                                           # inserir item na posição criada
```

- O laço externo (linha 4) é executado $n - 1$ vezes.
- E o laço interno?
  - **Melhor caso**: laço interno nunca é executado.
    - Em que situação isso ocorre? *Arranjo em ordem crescente.*
  - **Pior caso**: laço interno é executado i vezes para cada iteração do laço externo.
    - Em que situação isso ocorre?

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — análise

```
4:  para i ← 1 até n − 1 faça                                  # percorrer parte não-ordenada
5:      key ← A[i]                       # 1o item da parte não-ordenada (a ser inserido)
6:      holePos ← i − 1                      # começar a procurar do final da parte ordenada
7:      enquanto holePos ⩾ 0 e compara(key, A[holePos]) < 0 faça
8:          A[holePos + 1] ← A[holePos]                       # deslocar elementos p/ frente
9:          holePos ← holePos − 1                                      # continuar procura
10:     A[holePos + 1] ← key                               # inserir item na posição criada
```

- O laço externo (linha 4) é executado $n - 1$ vezes.
- E o laço interno?
  - **Melhor caso**: laço interno nunca é executado.
    - Em que situação isso ocorre? *Arranjo em ordem crescente.*
  - **Pior caso**: laço interno é executado i vezes para cada iteração do laço externo.
    - Em que situação isso ocorre? *Arranjo em ordem decrescente.*

- No pior caso laço interno é executado i vezes

$$T_{pior}(n) = \sum_{i=1}^{n-1}(i) = 1 + 2 + \cdots + n - 1$$

- No pior caso laço interno é executado i vezes

$$T_{pior}(n) = \sum_{i=1}^{n-1}(i) = 1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2} = O(n^2).$$

- No pior caso laço interno é executado i vezes

$$T_{pior}(n) = \sum_{i=1}^{n-1}(i) = 1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2} = O(n^2).$$

- No melhor caso $T(n) = \Omega(n)$.

- No pior caso laço interno é executado i vezes

$$T_{pior}(n) = \sum_{i=1}^{n-1}(i) = 1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2} = O(n^2).$$

- No melhor caso $T(n) = \Omega(n)$.
- Sua complexidade de pior caso (e caso médio) torna o *insertion sort* impraticável para ordenar vetores com muitos elementos.

- No pior caso laço interno é executado i vezes

$$T_{pior}(n) = \sum_{i=1}^{n-1}(i) = 1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2} = O(n^2).$$

- No melhor caso $T(n) = \Omega(n)$.
- Sua complexidade de pior caso (e caso médio) torna o *insertion sort* impraticável para ordenar vetores com muitos elementos.
- Apesar disso, é um dos algoritmos mais rápidos para um vetor pequeno, sendo até mais rápido do que o *quicksort*;

- No pior caso laço interno é executado i vezes

$$T_{pior}(n) = \sum_{i=1}^{n-1}(i) = 1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2} = O(n^2).$$

- No melhor caso $T(n) = \Omega(n)$.
- Sua complexidade de pior caso (e caso médio) torna o *insertion sort* impraticável para ordenar vetores com muitos elementos.
- Apesar disso, é um dos algoritmos mais rápidos para um vetor pequeno, sendo até mais rápido do que o *quicksort*; de fato, algumas implementações do *quicksort* utilizam o *insertion sort* quanto o tamanho do vetor fica abaixo de um certo limite (encontrado experimentalmente).

- O *insertion sort* termina para todas as entradas válidas?

- O *insertion sort* termina para todas as entradas válidas?
- Se sim, ele satisfaz as condições para o problema de ordenação?

- O *insertion sort* termina para todas as entradas válidas?
- Se sim, ele satisfaz as condições para o problema de ordenação?
  - Contém uma permutação do valor inicial de A.

- O *insertion sort* termina para todas as entradas válidas?
- Se sim, ele satisfaz as condições para o problema de ordenação?
  - Contém uma permutação do valor inicial de A.
  - A está ordenado: $A[0] \leqslant A[1] \leqslant \cdots \leqslant A[n-1]$.

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

- O *insertion sort* termina para todas as entradas válidas?
- Se sim, ele satisfaz as condições para o problema de ordenação?
  - Contém uma permutação do valor inicial de A.
  - A está ordenado: $A[0] \leqslant A[1] \leqslant \cdots \leqslant A[n-1]$.
- Porém, precisamos de uma prova formal de corretude!

- O *insertion sort* termina para todas as entradas válidas?
- Se sim, ele satisfaz as condições para o problema de ordenação?
  - Contém uma permutação do valor inicial de A.
  - A está ordenado: $A[0] \leqslant A[1] \leqslant \cdots \leqslant A[n-1]$.
- Porém, precisamos de uma prova formal de corretude!
- Devemos determinar o invariante de laço para provar corretude parcial e função decremento para provar término.

- O *insertion sort* termina para todas as entradas válidas?
- Se sim, ele satisfaz as condições para o problema de ordenação?
  - ⋆ Contém uma permutação do valor inicial de A.
  - ⋆ A está ordenado: $A[0] \leqslant A[1] \leqslant \cdots \leqslant A[n-1]$.
- Porém, precisamos de uma prova formal de corretude!
- Devemos determinar o invariante de laço para provar corretude parcial e função decremento para provar término.
- Lembre-se que o invariante de laço deve ser verdadeiro:

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

- O *insertion sort* termina para todas as entradas válidas?
- Se sim, ele satisfaz as condições para o problema de ordenação?
  - Contém uma permutação do valor inicial de A.
  - A está ordenado: $A[0] \leqslant A[1] \leqslant \cdots \leqslant A[n-1]$.
- Porém, precisamos de uma prova formal de corretude!
- Devemos determinar o invariante de laço para provar corretude parcial e função decremento para provar término.
- Lembre-se que o invariante de laço deve ser verdadeiro:
  - Antes do início do laço;

- O *insertion sort* termina para todas as entradas válidas?
- Se sim, ele satisfaz as condições para o problema de ordenação?
  - Contém uma permutação do valor inicial de A.
  - A está ordenado: $A[0] \leqslant A[1] \leqslant \cdots \leqslant A[n-1]$.
- Porém, precisamos de uma prova formal de corretude!
- Devemos determinar o invariante de laço para provar corretude parcial e função decremento para provar término.
- Lembre-se que o invariante de laço deve ser verdadeiro:
  - Antes do início do laço;
  - Durante a manutenção do laço; e

- O *insertion sort* termina para todas as entradas válidas?
- Se sim, ele satisfaz as condições para o problema de ordenação?
  - Contém uma permutação do valor inicial de A.
  - A está ordenado: $A[0] \leqslant A[1] \leqslant \cdots \leqslant A[n-1]$.
- Porém, precisamos de uma prova formal de corretude!
- Devemos determinar o invariante de laço para provar corretude parcial e função decremento para provar término.
- Lembre-se que o invariante de laço deve ser verdadeiro:
  - Antes do início do laço;
  - Durante a manutenção do laço; e
  - Na saída do laço.

- O *insertion sort* termina para todas as entradas válidas?
- Se sim, ele satisfaz as condições para o problema de ordenação?
  - Contém uma permutação do valor inicial de A.
  - A está ordenado: $A[0] \leqslant A[1] \leqslant \cdots \leqslant A[n-1]$.
- Porém, precisamos de uma prova formal de corretude!
- Devemos determinar o invariante de laço para provar corretude parcial e função decremento para provar término.
- Lembre-se que o invariante de laço deve ser verdadeiro:
  - Antes do início do laço;
  - Durante a manutenção do laço; e
  - Na saída do laço.
- Pré-condição: Arranho A com $n \geqslant 0$ elementos que admitem uma ordem total.

- O *insertion sort* termina para todas as entradas válidas?
- Se sim, ele satisfaz as condições para o problema de ordenação?
  - Contém uma permutação do valor inicial de A.
  - A está ordenado: $A[0] \leqslant A[1] \leqslant \cdots \leqslant A[n-1]$.
- Porém, precisamos de uma prova formal de corretude!
- Devemos determinar o invariante de laço para provar corretude parcial e função decremento para provar término.
- Lembre-se que o invariante de laço deve ser verdadeiro:
  - Antes do início do laço;
  - Durante a manutenção do laço; e
  - Na saída do laço.
- Pré-condição: Arranho A com $n \geqslant 0$ elementos que admitem uma ordem total.
- Pós-condição: Permutação A_π com $n \geqslant 0$ elementos ordenados.

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

para $i \leftarrow 1$ **até** $n-1$ **faça** `# percorrer parte não-ordenada`
    $key \leftarrow A[i]$ `# 1o item da parte não-ordenada (a ser inserido)`
    $holePos \leftarrow i-1$ `# começar a procurar do final da parte ordenada`
    enquanto $holePos \geqslant 0$ **e** `compara`$(key, A[holePos]) = -1$ **faça**
        $A[holePos+1] \leftarrow A[holePos]$ `# deslocar elementos p/ frente`
        $holePos \leftarrow holePos - 1$ `# continuar procura`
    $A[holePos+1] \leftarrow key$ `# inserir item na posição criada`

- A principal ideia é inserir $A[i]$ (i.e. key) em $A[0 \ldots i-1]$ de maneira a manter uma subsequência ordenada $A[0 \ldots i]$.

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

```
para i ← 1 até n − 1 faça                                  # percorrer parte não-ordenada
    key ← A[i]                                 # 1o item da parte não-ordenada (a ser inserido)
    holePos ← i − 1                              # começar a procurar do final da parte ordenada
    enquanto holePos ⩾ 0 e compara(key, A[holePos]) = −1 faça
        A[holePos + 1] ← A[holePos]                          # deslocar elementos p/ frente
        holePos ← holePos − 1                                            # continuar procura
    A[holePos + 1] ← key                                      # inserir item na posição criada
```

- ▷ A principal ideia é inserir $A[i]$ (i.e. key) em $A[0 \ldots i-1]$ de maneira a manter uma subsequência ordenada $A[0 \ldots i]$.
- ▷ **Invariante** (do laço externo):

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

para $i \leftarrow 1$ **até** $n-1$ **faça** `# percorrer parte não-ordenada`
  $key \leftarrow A[i]$ `# 1o item da parte não-ordenada (a ser inserido)`
  $holePos \leftarrow i-1$ `# começar a procurar do final da parte ordenada`
  enquanto $holePos \geqslant 0$ **e** `compara`$(key, A[holePos]) = -1$ **faça**
    $A[holePos+1] \leftarrow A[holePos]$ `# deslocar elementos p/ frente`
    $holePos \leftarrow holePos - 1$ `# continuar procura`
  $A[holePos+1] \leftarrow key$ `# inserir item na posição criada`

- A principal ideia é inserir $A[i]$ (i.e. key) em $A[0 \ldots i-1]$ de maneira a manter uma subsequência ordenada $A[0 \ldots i]$.
- **Invariante** (do laço externo): *o subarranjo $A[0 \ldots i-1]$ é uma permutação ordenada do subarranjo original $A[0 \ldots i-1]$.*

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

```
para i ← 1 até n − 1 faça                                              # percorrer parte não-ordenada
    key ← A[i]                                               # 1o item da parte não-ordenada (a ser inserido)
    holePos ← i − 1                                            # começar a procurar do final da parte ordenada
    enquanto holePos ⩾ 0 e compara(key, A[holePos]) = −1 faça
        A[holePos + 1] ← A[holePos]                                   # deslocar elementos p/ frente
        holePos ← holePos − 1                                                 # continuar procura
    A[holePos + 1] ← key                                              # inserir item na posição criada
```

▷ **Início**: $i \leftarrow 1$, de maneira que $A[0 \ldots i-1]$ é o único elemento $A[0]$.

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

```
para i ← 1 até n − 1 faça                                        # percorrer parte não-ordenada
    key ← A[i]                                  # 1o item da parte não-ordenada (a ser inserido)
    holePos ← i − 1                               # começar a procurar do final da parte ordenada
    enquanto holePos ⩾ 0 e compara(key, A[holePos]) = −1 faça
        A[holePos + 1] ← A[holePos]                             # deslocar elementos p/ frente
        holePos ← holePos − 1                                               # continuar procura
    A[holePos + 1] ← key                                        # inserir item na posição criada
```

- **Início**: $i \leftarrow 1$, de maneira que $A[0 \ldots i-1]$ é o único elemento $A[0]$.
  - ⋆ $A[0]$ contém o elemento original em $A[0]$.

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

```
para i ← 1 até n − 1 faça                                          # percorrer parte não-ordenada
    key ← A[i]                                         # 1o item da parte não-ordenada (a ser inserido)
    holePos ← i − 1                                     # começar a procurar do final da parte ordenada
    enquanto holePos ⩾ 0 e compara(key, A[holePos]) = −1 faça
        A[holePos + 1] ← A[holePos]                              # deslocar elementos p/ frente
        holePos ← holePos − 1                                                # continuar procura
    A[holePos + 1] ← key                                         # inserir item na posição criada
```

▷ **Início**: $i \leftarrow 1$, de maneira que $A[0 \dots i-1]$ é o único elemento $A[0]$.

- ⋆ $A[0]$ contém o elemento original em $A[0]$.
- ⋆ $A[0]$ é trivialmente ordenado.

```
para i ← 1 até n − 1 faça                                        # percorrer parte não-ordenada
    key ← A[i]                                     # 1o item da parte não-ordenada (a ser inserido)
    holePos ← i − 1                                  # começar a procurar do final da parte ordenada
    enquanto holePos ⩾ 0 e compara(key, A[holePos]) = −1 faça
        A[holePos + 1] ← A[holePos]                                # deslocar elementos p/ frente
        holePos ← holePos − 1                                                  # continuar procura
    A[holePos + 1] ← key                                           # inserir item na posição criada
```

▷ **Variação**: informalmente, se $A[0\ldots i-1]$ é uma permutação do subarranjo original $A[0\ldots i-1]$ e $A[0\ldots i-1]$ está ordenado (invariante), então *se* entramos no laço mais interno:

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

```
para i ← 1 até n − 1 faça                                      # percorrer parte não-ordenada
    key ← A[i]                                   # 1o item da parte não-ordenada (a ser inserido)
    holePos ← i − 1                                 # começar a procurar do final da parte ordenada
    enquanto holePos ⩾ 0 e compara(key, A[holePos]) = −1 faça
        A[holePos + 1] ← A[holePos]                              # deslocar elementos p/ frente
        holePos ← holePos − 1                                              # continuar procura
    A[holePos + 1] ← key                                          # inserir item na posição criada
```

- **Variação**: informalmente, se $A[0 \ldots i-1]$ é uma permutação do subarranjo original $A[0 \ldots i-1]$ e $A[0 \ldots i-1]$ está ordenado (invariante), então *se* entramos no laço mais interno:
  - ⋆ deslocamos $A[holePos \ldots i-1]$ de uma posição para a direita.

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

```
para i ← 1 até n − 1 faça                              # percorrer parte não-ordenada
    key ← A[i]                             # 1o item da parte não-ordenada (a ser inserido)
    holePos ← i − 1                          # começar a procurar do final da parte ordenada
    enquanto holePos ⩾ 0 e compara(key, A[holePos]) = −1 faça
        A[holePos + 1] ← A[holePos]                      # deslocar elementos p/ frente
        holePos ← holePos − 1                                     # continuar procura
    A[holePos + 1] ← key                                # inserir item na posição criada
```

▷ **Variação**: informalmente, se $A[0 \ldots i-1]$ é uma permutação do subarranjo original $A[0 \ldots i-1]$ e $A[0 \ldots i-1]$ está ordenado (invariante), então *se* entramos no laço mais interno:

- ⋆ deslocamos $A[holePos \ldots i-1]$ de uma posição para a direita.
- ⋆ inserimos *key*, que estava originalmente em $A[i]$, em sua prosição apropriada $0 \leqslant holePos \leqslant i-1$, de maneira a manter a ordenação (invariante).

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

```
para i ← 1 até n − 1 faça                                    # percorrer parte não-ordenada
    key ← A[i]                          # 1o item da parte não-ordenada (a ser inserido)
    holePos ← i − 1                       # começar a procurar do final da parte ordenada
    enquanto holePos ⩾ 0 e compara(key, A[holePos]) = −1 faça
        A[holePos + 1] ← A[holePos]                        # deslocar elementos p/ frente
        holePos ← holePos − 1                                        # continuar procura
    A[holePos + 1] ← key                                   # inserir item na posição criada
```

- **Progresso**: a cada iteração do laço (mais externo), a variávei i é incrementada em uma unidade.

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

```
para i ← 1 até n − 1 faça                                   #percorrer parte não-ordenada
    key ← A[i]                                    #1o item da parte não-ordenada (a ser inserido)
    holePos ← i − 1                                    #começar a procurar do final da parte ordenada
    enquanto holePos ≥ 0 e compara(key, A[holePos]) = −1 faça
        A[holePos + 1] ← A[holePos]                            #deslocar elementos p/ frente
        holePos ← holePos − 1                                          #continuar procura
    A[holePos + 1] ← key                                       #inserir item na posição criada
```

- **Progresso**: a cada iteração do laço (mais externo), a variávei i é incrementada em uma unidade.
- Portanto o variante ou função decremento $D(X) = n - i$ diminui a cada iteração.

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

```
para i ← 1 até n − 1 faça                                        #percorrer parte não-ordenada
    key ← A[i]                                    #1o item da parte não-ordenada (a ser inserido)
    holePos ← i − 1                                 #começar a procurar do final da parte ordenada
    enquanto holePos ⩾ 0 e compara(key, A[holePos]) = −1 faça
        A[holePos + 1] ← A[holePos]                                #deslocar elementos p/ frente
        holePos ← holePos − 1                                                 #continuar procura
    A[holePos + 1] ← key                                         #inserir item na posição criada
```

▷ **Limitação**: `insertionSort` termina com $i = n$; a invariante indica:

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

```
para i ← 1 até n − 1 faça                                      # percorrer parte não-ordenada
    key ← A[i]                                # 1o item da parte não-ordenada (a ser inserido)
    holePos ← i − 1                              # começar a procurar do final da parte ordenada
    enquanto holePos ⩾ 0 e compara(key, A[holePos]) = −1 faça
        A[holePos + 1] ← A[holePos]                                # deslocar elementos p/ frente
        holePos ← holePos − 1                                                # continuar procura
    A[holePos + 1] ← key                                          # inserir item na posição criada
```

- **Limitação**: `insertionSort` termina com $i = n$; a invariante indica:
  - ⋆ $A[0 \ldots i-1]$ é uma permutação do subarranjo original $A[0 \ldots i-1]$.

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

```
para i ← 1 até n − 1 faça                                   # percorrer parte não-ordenada
    key ← A[i]                            # 1o item da parte não-ordenada (a ser inserido)
    holePos ← i − 1                           # começar a procurar do final da parte ordenada
    enquanto holePos ⩾ 0 e compara(key, A[holePos]) = −1 faça
        A[holePos + 1] ← A[holePos]                          # deslocar elementos p/ frente
        holePos ← holePos − 1                                          # continuar procura
    A[holePos + 1] ← key                                     # inserir item na posição criada
```

▷ **Limitação**: `insertionSort` termina com $i = n$; a invariante indica:

- ⋆ $A[0 \ldots i-1]$ é uma permutação do subarranjo original $A[0 \ldots i-1]$.
- ⋆ $A[0 \ldots i-1]$ está ordenado.

Ordenação por Inserção

(2) Algoritmo **Insertion Sort** — corretude

```
para i ← 1 até n − 1 faça                                  # percorrer parte não-ordenada
    key ← A[i]                               # 1o item da parte não-ordenada (a ser inserido)
    holePos ← i − 1                          # começar a procurar do final da parte ordenada
    enquanto holePos ⩾ 0 e compara(key, A[holePos]) = −1 faça
        A[holePos + 1] ← A[holePos]                         # deslocar elementos p/ frente
        holePos ← holePos − 1                                # continuar procura
    A[holePos + 1] ← key                                    # inserir item na posição criada
```

- ▻ **Limitação**: `insertionSort` termina com $i = n$; a invariante indica:
  - ⋆ $A[0 \ldots i-1]$ é uma permutação do subarranjo original $A[0 \ldots i-1]$.
  - ⋆ $A[0 \ldots i-1]$ está ordenado.
- ▻ Dado a condição de término, $A[0 \ldots i-1]$ corresponde a todo o arranho A; logo, `insertionSort` é **correto**! $\square$

- O melhor e o pior casos do *insertion sort* ocorrem quando A está ordenado em ordem crescente e decrescente, respectivamente.

- O melhor e o pior casos do *insertion sort* ocorrem quando A está ordenado em ordem crescente e decrescente, respectivamente.
  - No melhor caso, o tempo de execução é $\Omega(n)$.

- O melhor e o pior casos do *insertion sort* ocorrem quando A está ordenado em ordem crescente e decrescente, respectivamente.
  - No melhor caso, o tempo de execução é $\Omega(n)$.
  - No pior caso, o tempo de execução é $O(n^2)$.

- O melhor e o pior casos do *insertion sort* ocorrem quando A está ordenado em ordem crescente e decrescente, respectivamente.
  - No melhor caso, o tempo de execução é $\Omega(n)$.
  - No pior caso, o tempo de execução é $O(n^2)$.
- É de fácil implementação.

- O melhor e o pior casos do *insertion sort* ocorrem quando A está ordenado em ordem crescente e decrescente, respectivamente.
  - No melhor caso, o tempo de execução é $\Omega(n)$.
  - No pior caso, o tempo de execução é $O(n^2)$.
- É de fácil implementação.
- É relativamente eficiente para ordenar poucos dados.

- O melhor e o pior casos do *insertion sort* ocorrem quando A está ordenado em ordem crescente e decrescente, respectivamente.
  - No melhor caso, o tempo de execução é $\Omega(n)$.
  - No pior caso, o tempo de execução é $O(n^2)$.
- É de fácil implementação.
- É relativamente eficiente para ordenar poucos dados.
- É adaptativo, ou seja, eficiente para dados ordenados.

- O melhor e o pior casos do *insertion sort* ocorrem quando A está ordenado em ordem crescente e decrescente, respectivamente.
  - No melhor caso, o tempo de execução é $\Omega(n)$.
  - No pior caso, o tempo de execução é $O(n^2)$.
- É de fácil implementação.
- É relativamente eficiente para ordenar poucos dados.
- É adaptativo, ou seja, eficiente para dados ordenados.
- É estável, visto que não altera a ordem relativa de elementos com chaves repetidas.

- O melhor e o pior casos do *insertion sort* ocorrem quando A está ordenado em ordem crescente e decrescente, respectivamente.
  - No melhor caso, o tempo de execução é $\Omega(n)$.
  - No pior caso, o tempo de execução é $O(n^2)$.
- É de fácil implementação.
- É relativamente eficiente para ordenar poucos dados.
- É adaptativo, ou seja, eficiente para dados ordenados.
- É estável, visto que não altera a ordem relativa de elementos com chaves repetidas.
- É *in-place*, já que apenas requer uma quantidade de memória constante $O(1)$ para funcionar.

- O melhor e o pior casos do *insertion sort* ocorrem quando A está ordenado em ordem crescente e decrescente, respectivamente.
  - ⋆ No melhor caso, o tempo de execução é $\Omega(n)$.
  - ⋆ No pior caso, o tempo de execução é $O(n^2)$.
- É de fácil implementação.
- É relativamente eficiente para ordenar poucos dados.
- É adaptativo, ou seja, eficiente para dados ordenados.
- É estável, visto que não altera a ordem relativa de elementos com chaves repetidas.
- É *in-place*, já que apenas requer uma quantidade de memória constante $O(1)$ para funcionar.
- É *online*, pois pode ordenar uma lista à medida que a recebe.

- O melhor e o pior casos do *insertion sort* ocorrem quando A está ordenado em ordem crescente e decrescente, respectivamente.
  - ⋆ No melhor caso, o tempo de execução é $\Omega(n)$.
  - ⋆ No pior caso, o tempo de execução é $O(n^2)$.
- É de fácil implementação.
- É relativamente eficiente para ordenar poucos dados.
- É adaptativo, ou seja, eficiente para dados ordenados.
- É estável, visto que não altera a ordem relativa de elementos com chaves repetidas.
- É *in-place*, já que apenas requer uma quantidade de memória constante $O(1)$ para funcionar.
- É *online*, pois pode ordenar uma lista à medida que a recebe.
- É correto, pois termina para entradas válidas com o arranjo ordenado.

- A principal desvantagem do *insertion sort* é que, para inserir um elemento na parte ordenada pode ser necessário deslocar muitos itens de um vetor ou manipular elementos de uma lista.

- A principal desvantagem do *insertion sort* é que, para inserir um elemento na parte ordenada pode ser necessário deslocar muitos itens de um vetor ou manipular elementos de uma lista.
- Deslocar pode ser custoso:

- A principal desvantagem do *insertion sort* é que, para inserir um elemento na parte ordenada pode ser necessário deslocar muitos itens de um vetor ou manipular elementos de uma lista.
- Deslocar pode ser custoso:
  - Elemento com um *footprint* de memória grande.

- A principal desvantagem do *insertion sort* é que, para inserir um elemento na parte ordenada pode ser necessário deslocar muitos itens de um vetor ou manipular elementos de uma lista.
- Deslocar pode ser custoso:
  - Elemento com um *footprint* de memória grande.
  - Vetor com muitos elementos.

- A principal desvantagem do *insertion sort* é que, para inserir um elemento na parte ordenada pode ser necessário deslocar muitos itens de um vetor ou manipular elementos de uma lista.
- Deslocar pode ser custoso:
  - Elemento com um *footprint* de memória grande.
  - Vetor com muitos elementos.
  - Vetor com elementos armazenados em dispositivos externos.

- A principal desvantagem do *insertion sort* é que, para inserir um elemento na parte ordenada pode ser necessário deslocar muitos itens de um vetor ou manipular elementos de uma lista.
- Deslocar pode ser custoso:
  - Elemento com um *footprint* de memória grande.
  - Vetor com muitos elementos.
  - Vetor com elementos armazenados em dispositivos externos.
- Assim, a motivação do *selection sort* é eliminar estes deslocamentos.

- Em linhas gerais o algoritmo funciona da seguinte forma:

- Em linhas gerais o algoritmo funciona da seguinte forma:
    1. Encontre o menor valor da coleção (lista/vetor).

- Em linhas gerais o algoritmo funciona da seguinte forma:
  1. Encontre o menor valor da coleção (lista/vetor).
  2. Troque-o com o valor na primeira posição.

- Em linhas gerais o algoritmo funciona da seguinte forma:

  1. Encontre o menor valor da coleção (lista/vetor).
  2. Troque-o com o valor na primeira posição.
  3. Repita os passos acima para o restante da lista (iniciando na segunda posição e assim avançando a cada iteração).

- Em linhas gerais o algoritmo funciona da seguinte forma:
  1. Encontre o menor valor da coleção (lista/vetor).
  2. Troque-o com o valor na primeira posição.
  3. Repita os passos acima para o restante da lista (iniciando na segunda posição e assim avançando a cada iteração).
- Na prática, a coleção é dividido em duas partes:

- Em linhas gerais o algoritmo funciona da seguinte forma:
  1. Encontre o menor valor da coleção (lista/vetor).
  2. Troque-o com o valor na primeira posição.
  3. Repita os passos acima para o restante da lista (iniciando na segunda posição e assim avançando a cada iteração).
- Na prática, a coleção é dividido em duas partes:
  - uma região de itens já ordenados, o qual é construído da esquerda para direita e localiza-se no início; e

- Em linhas gerais o algoritmo funciona da seguinte forma:

  1. Encontre o menor valor da coleção (lista/vetor).
  2. Troque-o com o valor na primeira posição.
  3. Repita os passos acima para o restante da lista (iniciando na segunda posição e assim avançando a cada iteração).

- Na prática, a coleção é dividido em duas partes:

  - uma região de itens já ordenados, o qual é construído da esquerda para direita e localiza-se no início; e
  - uma região de itens que precisam ser ordenados, ocupando o restante da coleção.

Iteração #0 $\Rightarrow A$: | 5 | 2 | 4 | 6 | 1 | 3 |

Iteração #1 ⇒ A: | 5 | 2 | 4 | 6 | 1 | 3 | Menor: | 1 |

Iteração #1 ⇒ A: | 1 | 2 | 4 | 6 | 5 | 3 | Menor: | 1 |

Iteração #1 ⇒ A: $\boxed{1}$ $\boxed{2}\boxed{4}\boxed{6}\boxed{5}\boxed{3}$

Iteração #1 $\Rightarrow A$: $\boxed{1}$ $\boxed{2}\boxed{4}\boxed{6}\boxed{5}\boxed{3}$

Iteração #2 $\Rightarrow A$: $\boxed{1}$ $\boxed{2}\boxed{4}\boxed{6}\boxed{5}\boxed{3}$ Menor: $\boxed{2}$

Iteração #2 ⇒ A:

1	2	4	6	5	3

Iteração #2 ⇒ A: $\boxed{1}\boxed{2}\ \ \boxed{4}\boxed{6}\boxed{5}\boxed{3}$

Iteração #3 $\Rightarrow A$: $\boxed{1}\boxed{2}$ $\boxed{4}\boxed{6}\boxed{5}\boxed{\color{red}{3}}$ Menor: $\boxed{3}$

Iteração #3 ⇒ A: | 1 | 2 | | 3 | 6 | 5 | 4 | Menor: | 3 |

Iteração #3 $\Rightarrow$ A: $\boxed{1}\boxed{2}\boxed{\color{red}3}\ \boxed{6}\boxed{5}\boxed{4}$

Iteração #3 $\Rightarrow A$: $\boxed{1}\boxed{2}\boxed{3}\;\boxed{6}\boxed{5}\boxed{4}$

Iteração #4 ⇒ A: $\boxed{1}\boxed{2}\boxed{3}$ $\boxed{6}\boxed{5}\boxed{4}$ Menor: $\boxed{4}$

Iteração #4 $\Rightarrow A$: $\boxed{1}\boxed{2}\boxed{3}$ $\boxed{4}\boxed{5}\boxed{6}$ Menor: $\boxed{4}$

Iteração #4 ⇒ A: | 1 | 2 | 3 | 4 | | 5 | 6 |

Iteração #4 $\Rightarrow A$: $\boxed{1}\boxed{2}\boxed{3}\boxed{4}\ \ \boxed{5}\boxed{6}$

Iteração #5 $\Rightarrow$ A: $\boxed{1}\boxed{2}\boxed{3}\boxed{4}\ \boxed{5}\boxed{6}$ Menor: $\boxed{5}$

Iteração #5 $\Rightarrow A$: $\boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{5}$ $\boxed{6}$ Menor: $\boxed{5}$

Iteração #5 $\Rightarrow A$: $\boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{5}\ \boxed{6}$

Iteração #6 $\Rightarrow$ A: | 1 | 2 | 3 | 4 | 5 | | 6 | Menor: | 6 |

Iteração #6 $\Rightarrow A$: | 1 | 2 | 3 | 4 | 5 | 6 | Menor: | 6 |

Iteração #6 ⇒ A:

1	2	3	4	5	6

- De maneira geral, na i-ésima passagem pela lista, que varia de 0 a $n-2$, o algoritmo busca pelo menor item entre os último $n-i$ elementos e o troca com A_i:

$$A_0 \leqslant A_1 \leqslant \cdots \leqslant A_{i-1} \mid A_i, \ldots, A_{min}, \ldots, A_{n-1}$$

- De maneira geral, na i-ésima passagem pela lista, que varia de 0 a $n-2$, o algoritmo busca pelo menor item entre os último $n-i$ elementos e o troca com A_i:

$$A_0 \leqslant A_1 \leqslant \cdots \leqslant A_{i-1} \mid A_i, \ldots, A_{min}, \ldots, A_{n-1}$$

- Depois de $n-1$ passagens, a lista está ordenada.

Selection Sort

procedimento `selectionSort`(A: **arranjo de ref inteiro**)
    var n: **inteiro** $\leftarrow$ **tam** A # `recuperar tamanho vetor`
    var $menor$, i, j: **inteiro**   # `variáveis auxiliares`
    para $i \leftarrow 0$ **até** $n-2$ **faça**   # `percorrer até penúltimo`
        $menor \leftarrow i$ # `guardar índice do atual menor`
        para $j \leftarrow i+1$ **até** $n-1$ **faça**   # `subvetor não ordenado`
            se `compara`($A[j]$, $A[menor]$)< 0 **então**
                $menor \leftarrow j$ # `atualizar menor`
        $A[menor] \leftrightarrow A[i]$ # `realizar a troca`

Ordenação por Seleção

(3) Algoritmo **Selection Sort — análise**

para $i \leftarrow 0$ **até** $n-2$ **faça** #percorrer até penúltimo
  $menor \leftarrow i$ #guardar índice do atual menor
  para $j \leftarrow i+1$ **até** $n-1$ **faça** #subvetor não ordenado
    se compara($A[j]$, $A[menor]$)< 0 **então**
      $menor \leftarrow j$ #atualizar menor
  $A[menor] \leftrightarrow A[i]$ #realizar a troca

▷ A entrada é definida em função do tamanho n do arranjo.

Ordenação por Seleção

(3) Algoritmo **Selection Sort — análise**

```
4: para i ← 0 até n − 2 faça                              # percorrer até penúltimo
5:     menor ← i                                          # guardar índice do atual menor
6:     para j ← i + 1 até n − 1 faça                      # subvetor não ordenado
7:         se compara(A[j], A[menor]) < 0 então
8:             menor ← j                                  # atualizar menor
9:     A[menor] ↔ A[i]                                    # realizar a troca
```

- A entrada é definida em função do tamanho n do arranjo.
- A operação dominante é a comparação de chaves (linha 7).

$$T(n) \leqslant \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \leqslant \sum_{i=0}^{n-2} [(n-1)-(i+1)+1] \leqslant \sum_{i=0}^{n-2} (n-1-i)$$

$$T(n) \leqslant \frac{n(n-1)}{2} \in \Theta(n^2).$$

```
para i ← 0 até n − 2 faça                                  # percorrer até penúltimo
    menor ← i                                               # guardar índice do atual menor
    para j ← i + 1 até n − 1 faça                           # subvetor não ordenado
        se compara(A[j], A[menor]) = −1 então
            menor ← j                                       # atualizar menor
    A[menor] ↔ A[i]                                         # realizar a troca
```

▷ O melhor caso do *selection sort* ocorre quando A está ordenado em ordem crescente.

Ordenação por Seleção

(3) Algoritmo **Selection Sort — análise**

```
4: para i ← 0 até n − 2 faça                                  # percorrer até penúltimo
5:     menor ← i                                              # guardar índice do atual menor
6:     para j ← i + 1 até n − 1 faça                          # subvetor não ordenado
7:         se compara(A[j], A[menor]) = −1 então
8:             menor ← j                                      # atualizar menor
9:     A[menor] ↔ A[i]                                        # realizar a troca
```

- O melhor caso do *selection sort* ocorre quando A está ordenado em ordem crescente.
- Já o pior caso ocorre quando A está em ordem decrescente, embora isso não seja óbvio — *Por que a linha 08 é executada um número máximo de vezes quando A está em ordem descrescente?*

Ordenação por Seleção

(3) Algoritmo **Selection Sort — análise**

```
4: para i ← 0 até n − 2 faça                              # percorrer até penúltimo
5:     menor ← i                                          # guardar índice do atual menor
6:     para j ← i + 1 até n − 1 faça                      # subvetor não ordenado
7:         se compara(A[j], A[menor]) = −1 então
8:             menor ← j                                  # atualizar menor
9:     A[menor] ↔ A[i]                                    # realizar a troca
```

- O melhor caso do *selection sort* ocorre quando A está ordenado em ordem crescente.
- Já o pior caso ocorre quando A está em ordem decrescente, embora isso não seja óbvio — *Por que a linha 08 é executada um número máximo de vezes quando A está em ordem descrescente?*
- No entanto, tanto no melhor quanto no pior caso, o tempo de execução é $\Theta(n^2)$, pois a linha 07 é executada $\Theta(n^2)$ vezes, não importando como os elementos a ordenar estão dispostos em A.

- O principal representante da estratégia de trocas sucessivas é o algoritmo *bubble sort*, sendo um dos mais fáceis de entender e implementar.

- O principal representante da estratégia de trocas sucessivas é o algoritmo *bubble sort*, sendo um dos mais fáceis de entender e implementar.
- Em linhas gerais o algoritmo funciona com a seguinte filosofia:

- O principal representante da estratégia de trocas sucessivas é o algoritmo *bubble sort*, sendo um dos mais fáceis de entender e implementar.
- Em linhas gerais o algoritmo funciona com a seguinte filosofia:

   1. Repetidamente percorre a lista a ser ordenada, comparando cada par de itens adjacentes, trocando-os se estiver na ordem errada.

- O principal representante da estratégia de trocas sucessivas é o algoritmo *bubble sort*, sendo um dos mais fáceis de entender e implementar.
- Em linhas gerais o algoritmo funciona com a seguinte filosofia:
  1. Repetidamente percorre a lista a ser ordenada, comparando cada par de itens adjacentes, trocando-os se estiver na ordem errada.
  2. O percorrimento da lista é repetido até que nenhuma troca é mais necessária, o que indica que a lista está ordenada.

- O principal representante da estratégia de trocas sucessivas é o algoritmo *bubble sort*, sendo um dos mais fáceis de entender e implementar.
- Em linhas gerais o algoritmo funciona com a seguinte filosofia:
  1. Repetidamente percorre a lista a ser ordenada, comparando cada par de itens adjacentes, trocando-os se estiver na ordem errada.
  2. O percorrimento da lista é repetido até que nenhuma troca é mais necessária, o que indica que a lista está ordenada.
- O nome "bolha" deriva do fato de que os menores (maiores) elementos "borbulham" rapidamente para a frente (final) da lista.

Iteração # 0 $\Rightarrow A$:

5	2	4	6	1	3

Iteração # 1 $\Rightarrow A$:

5	2	4	6	1	3

Iteração # 1 $\Rightarrow A$:

2	5	4	6	1	3

Iteração # 1 $\Rightarrow A$:

2	5	4	6	1	3

Iteração # 1 $\Rightarrow A$:

2	4	5	6	1	3

Iteração # 1 $\Rightarrow A$:

2	4	5	6	1	3

Iteração # 1 $\Rightarrow A$:

2	4	5	6	1	3

Iteração # 1 $\Rightarrow A$:

2	4	5	1	6	3

Iteração # 1 $\Rightarrow A$:

2	4	5	1	6	3

Iteração # 1 $\Rightarrow A$:

2	4	5	1	3	6

2	4	5	1	3	6

Iteração # 2 $\Rightarrow A$:

2	4	5	1	3	6

2	4	5	1	3	6

Iteração # 2 $\Rightarrow A$:

Ordenação por Troca

(4) Algoritmo **Bubble Sort** — exemplo

2	4	5	1	3	6

2	4	5	1	3	6

2	4	5	1	3	6

Iteração # 2 $\Rightarrow A$:

Ordenação por Troca

(4) Algoritmo **Bubble Sort** — exemplo

2	4	5	1	3	6

2	4	5	1	3	6

2	4	5	1	3	6

Iteração # 2 $\Rightarrow A$:

2	4	1	5	3	6

Ordenação por Troca

(4) Algoritmo **Bubble Sort** — exemplo

Iteração # 2 $\Rightarrow A$:

2	4	5	1	3	6

2	4	5	1	3	6

2	4	5	1	3	6

2	4	1	5	3	6

2	4	1	5	3	6

Ordenação por Troca

(4) Algoritmo **Bubble Sort** — exemplo

Iteração # 2 $\Rightarrow A$:

2	4	5	1	3	6
2	4	5	1	3	6
2	4	5	1	3	6
2	4	1	5	3	6
2	4	1	5	3	6
2	4	1	3	5	6

Ordenação por Troca

(4) Algoritmo **Bubble Sort** — exemplo

Iteração # 2 $\Rightarrow A$:

2	4	5	1	3	6

2	4	5	1	3	6

2	4	5	1	3	6

2	4	1	5	3	6

2	4	1	5	3	6

2	4	1	3	5	6

2	4	1	3	5	6

	2	4	1	3	5	6
	2	4	1	3	5	6
	2	1	4	3	5	6
Iteração # 3 $\Rightarrow A$:	2	1	4	3	5	6
	2	1	3	4	5	6
	2	1	3	4	5	6
	2	1	3	4	5	6

Ordenação por Troca

(4) Algoritmo **Bubble Sort** — exemplo

Iteração # 4 $\Rightarrow A$:

2	1	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6

Iteração # 5 $\Rightarrow A$:

1	2	3	4	5	6

1	2	3	4	5	6

1	2	3	4	5	6

1	2	3	4	5	6

1	2	3	4	5	6

Sem trocas nesta iteração. Fim do algoritmo!

Bubble Sort (versão clássica)

procedimento `bubbleSortCla`(A: **arranjo de ref inteiro**)

  var n: **inteiro** $\leftarrow$ **tam** A # `recuperar tamanho vetor`

  var i, j: **inteiro** # `variáveis auxiliares`

  para $j \leftarrow 0$ **até** $n-1$ **faça** # `realizar trocas` n `vezes`

    para $i \leftarrow 0$ **até** $n-2$ **faça** # `tentar` $n-1$ `trocas`

      se `compara`($A[i+1]$, $A[i]$)< 0 **então**

        $A[i] \leftrightarrow A[i+1]$ # `realizar a troca`

Bubble Sort (versão otimizada #1)

```
procedimento bubbleSortOpt1(A: arranjo de ref inteiro)
    var n: inteiro ← tam A                          # recuperar tamanho vetor
    var i: inteiro                                  # variável auxiliar
    var houverTroca: booleano ← falso               # indica se houve troca
    repita                      # percorrer novamente se houve alguma troca
        houverTroca ← falso
        para i ← 0 até n − 2 faça                   # percorrer até penúltimo
            se compara(A[i + 1], A[i]) < 0 então
                A[i] ↔ A[i + 1]                     # realizar a troca
                houverTroca ← verdadeiro            # indicar troca
    até não houverTroca
```

Ordenação por Troca

Bubble Sort (versão otimizada #2)

procedimento `bubbleSortOpt2`(A: **arranjo de ref inteiro**)
  var n: **inteiro** $\leftarrow$ **tam** A # `recuperar tamanho vetor`
  var i, j: **inteiro** # `variáveis auxiliares`
  var $houverTroca$: **booleano** $\leftarrow$ falso # `indica se houve troca`
  $j \leftarrow n - 2$ # `tamanho inicial do vetor, -1 elemento`
  repita # `percorrer novamente se houve alguma troca`
    $houverTroca \leftarrow$ falso
    para $i \leftarrow 0$ **até** j **faça** # `percorrer até penúltimo “válido”`
      se `compara`$(A[i+1], A[i]) < 0$ **então**
        $A[i] \leftrightarrow A[i+1]$ # `realizar a troca`
        $houverTroca \leftarrow$ verdadeiro # `indicar troca`
    $j \leftarrow j - 1$ # `cada iteração temos 1 elemento na posição final`
  até não $houverTroca$

```
para j ← 0 até n − 1 faça                          # realizar trocas n vezes
    para i ← 0 até n − 2 faça                      # tentar n − 1 trocas
        se compara(A[i + 1], A[i]) < 0 então
            A[i] ↔ A[i + 1]                        # realizar a troca
```

- A entrada é definida em função do tamanho n do arranho.

Ordenação por Troca

(4) Algoritmo **Bubble Sort** — análise versão clássica

```
4: para j ← 0 até n − 1 faça                     #realizar trocas n vezes
5:     para i ← 0 até n − 2 faça                  #tentar n − 1 trocas
6:         se compara(A[i + 1], A[i]) < 0 então
7:             A[i] ↔ A[i + 1]                     #realizar a troca
```

- A entrada é definida em função do tamanho n do arranho.
- A operação básica é a comparação de chaves (linha 6 do alg. clássico).

$$T(n) = \sum_{j=0}^{n-1} \sum_{i=0}^{n-2} 1 = \sum_{j=0}^{n-1} [(n-2) - 0 + 1]$$

```
4: para j ← 0 até n − 1 faça                                  #realizar trocas n vezes
5:     para i ← 0 até n − 2 faça                               #tentar n − 1 trocas
6:         se compara(A[i + 1], A[i]) < 0 então
7:             A[i] ↔ A[i + 1]                                  #realizar a troca
```

- A entrada é definida em função do tamanho n do arranho.
- A operação básica é a comparação de chaves (linha 6 do alg. clássico).

$$T(n) = \sum_{j=0}^{n-1} \sum_{i=0}^{n-2} 1 = \sum_{j=0}^{n-1} [(n-2) - 0 + 1]$$

$$T(n) = \sum_{j=0}^{n-1} (n-1) = n(n-1) \in \Theta(n^2).$$

- O melhor e pior casos do *bubble sort* ocorrem quando A está em ordem crescente e decrescente, respectivamente.

- O melhor e pior casos do *bubble sort* ocorrem quando A está em ordem crescente e decrescente, respectivamente.
- No entanto, tanto no melhor como no pior caso, o tempo de execução (clássico) é $\Theta(n^2)$, pois a linha 06 (operação dominante) é executada $\Theta(n^2)$ vezes, não importando como os elementos a serem ordendos estão dispostos em A.

- O melhor e pior casos do *bubble sort* ocorrem quando A está em ordem crescente e decrescente, respectivamente.
- No entanto, tanto no melhor como no pior caso, o tempo de execução (clássico) é $\Theta(n^2)$, pois a linha 06 (operação dominante) é executada $\Theta(n^2)$ vezes, não importando como os elementos a serem ordendos estão dispostos em A.
- A versão otimizada do algoritmo apresenta comportamento linear no melhor caso

- O melhor e pior casos do *bubble sort* ocorrem quando A está em ordem crescente e decrescente, respectivamente.
- No entanto, tanto no melhor como no pior caso, o tempo de execução (clássico) é $\Theta(n^2)$, pois a linha 06 (operação dominante) é executada $\Theta(n^2)$ vezes, não importando como os elementos a serem ordendos estão dispostos em A.
- A versão otimizada do algoritmo apresenta comportamento linear no melhor caso — *Por que?*

Considerações Finais

- Os algoritmos que acabamos de estudar não são eficientes e, por isso, são raramente utilizados na prática.

Considerações Finais

- Os algoritmos que acabamos de estudar não são eficientes e, por isso, são raramente utilizados na prática.
- O maior problema com eles não está na complexidade do pior caso (que é $\Theta(n^2)$), mas sim no fato da complexidade de caso médio ser também quadrática!

Considerações Finais

- Os algoritmos que acabamos de estudar não são eficientes e, por isso, são raramente utilizados na prática.
- O maior problema com eles não está na complexidade do pior caso (que é $\Theta(n^2)$), mas sim no fato da complexidade de caso médio ser também quadrática!
- A seguir, estudaremos algoritmos mais eficientes, que possuem complexidade de caso médio $\Theta(n \lg n)$.

Considerações Finais

- Os algoritmos que acabamos de estudar não são eficientes e, por isso, são raramente utilizados na prática.
- O maior problema com eles não está na complexidade do pior caso (que é $\Theta(n^2)$), mas sim no fato da complexidade de caso médio ser também quadrática!
- A seguir, estudaremos algoritmos mais eficientes, que possuem complexidade de caso médio $\Theta(n \lg n)$.
- Alguns destes, inclusive, possuem complexidade de pior caso $\Theta(n \lg n)$ também!

Considerações Finais

- Os algoritmos que acabamos de estudar não são eficientes e, por isso, são raramente utilizados na prática.
- O maior problema com eles não está na complexidade do pior caso (que é $\Theta(n^2)$), mas sim no fato da complexidade de caso médio ser também quadrática!
- A seguir, estudaremos algoritmos mais eficientes, que possuem complexidade de caso médio $\Theta(n \lg n)$.
- Alguns destes, inclusive, possuem complexidade de pior caso $\Theta(n \lg n)$ também!
- Estes últimos são considerados **ótimos** para o problema de ordenação usando comparação de chaves.

Paulo A. Azeredo
Métodos de Classificação de Dados,
Editora Campus, 1996.

Robert L. Kruse, Alexander J. Ryba
Data Structures and Program Design in C++, Third Edition, **Cap. 8**.
Prentice Hall, New Jersey/USA, Addison Wesley, 2000.