Algoritmos e Estruturas de Dados I - DIM0110.0

Selan R. dos Santos

DIMAp – Departamento de Informática e Matemática Aplicada
Sala 25, ramal 239, selan@dimap.ufrn.br
UFRN

2010.2

Algoritmos de Ordenação — Conteúdo

Introdução

- 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

Introdução

- 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 processamnto dos computadores mordernos e seus algoritmos

Introdução

- 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 processamnto 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

Introdução

- 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 processamnto 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

Introdução

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

Introdução

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

- 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

Introdução

- 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.

Introdução

- 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 $a = b$;

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 $a = b$;
  - **Transitividade:**
    se $a \leqslant b$ e $b \leqslant c$ então $a \leqslant c$;

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 $a = b$;
  - **Transitividade:**
    se $a \leqslant b$ e $b \leqslant c$ então $a \leqslant c$;
  - **Totalidade:**
    $a \leqslant b$ ou $b \leqslant a$.

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 $a \in X$)
- Se uma relação possui esta propriedade, então quaisquer dois elementos do conjunto sobre o qual a relação está definida 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:

$$a < b \text{ se e somente se } a \leqslant b \text{ e } a \neq b,\ \forall a, b \in X$$

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:

$$a < b \text{ se e somente se } a \leqslant b \text{ e } a \neq b, \; \forall a, b \in X$$

- A relação $<$ é transitiva

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:

$$a < b \text{ se e somente se } a \leqslant b \text{ e } a \neq b,\ \forall a, b \in X$$

- A relação $<$ é transitiva
- Além disso, para quaisquer $a, b \in X$, apenas uma das três possibilidades ocorre:

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:

$$a < b \text{ se e somente se } a \leqslant b \text{ e } a \neq b,\ \forall a, b \in X$$

- 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:

$$a < b \text{ se e somente se } a \leqslant b \text{ e } a \neq b, \; \forall a, b \in X$$

- 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:

$$a < b \text{ se e somente se } a \leqslant b \text{ e } a \neq b,\ \forall a, b \in X$$

- A relação $<$ é transitiva
- Além disso, para quaisquer $a, b \in X$, apenas uma das três possibilidades ocorre:
  - ⋆ $a < b$;
  - ⋆ $b < a$; ou
  - ⋆ $a = b$.

Definição Ordem Total e Ordem Total Estrita

- 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. $A < B < C < D < \cdots < Z$

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"*

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

O Problema de Ordenação

Entrada:

Uma sequência,

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

de n elementos, com $n \in \mathbb{Z}$ e $n > 0$, 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 $n \in \mathbb{Z}$ e $n > 0$, 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 $\leqslant$.

- Vamos estudar algoritmos para resolver o problema 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

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

- 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

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

Ordenação por Inserção

- Assim, algoritmos 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 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:

Ordenação por Inserção

- Assim, algoritmos 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 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 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

Com memória sequencial (a.k.a. vetor)

- É necessário deslocar elementos para dar espaço para a inserção

Com memória sequencial (a.k.a. vetor)

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

Com memória sequencial (a.k.a. vetor)

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

Com memória sequencial (a.k.a. vetor)

- 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. vetor)

- 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. vetor)

- 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. vetor)

- 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. vetor)

- ▻ 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. vetor)

- 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. vetor)

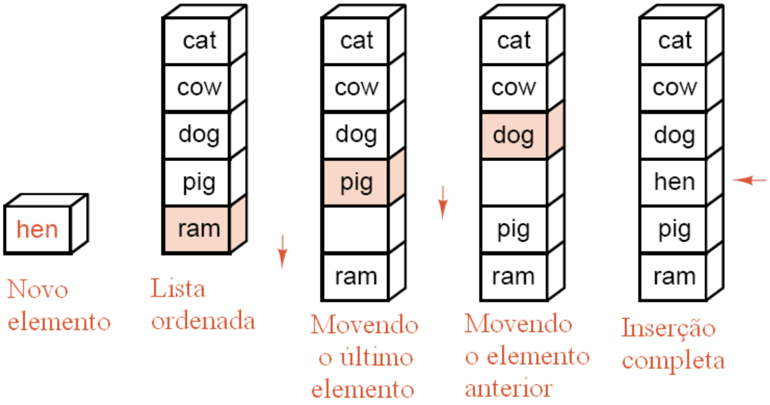
- É necessário deslocar elementos para dar espaço para a inserção
- Procedimento:
  1. Encontrar o local de inserção por:
     - Método #1: busca binária
     - Método #2: busca sequencial a partir do fim do vetor
  2. Deslocar os elementos seguintes ao local de inserção
  3. Inserir novo elemento no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

Ordenação por Inserção

(1) Algoritmo Inserção Ordenada — exemplo



- Ordenação a partir de uma estrutura ordenada ou não-ordenada

- Ordenação a partir de uma estrutura ordenada ou não-ordenada
- Separa a estrutura a ser ordenada em duas partes: já ordenada e não-ordenada

- Ordenação a partir de uma estrutura ordenada ou não-ordenada
- Separa a estrutura a ser ordenada em duas partes: já ordenada e não-ordenada
- Procedimento:

- Ordenação a partir de uma estrutura ordenada ou não-ordenada
- Separa a estrutura a ser ordenada em duas partes: já ordenada e não-ordenada
- Procedimento:
    1. Para todo item da parte não-ordenada faça:

- Ordenação a partir de uma estrutura ordenada ou não-ordenada
- Separa a estrutura a ser ordenada em duas partes: já ordenada e não-ordenada
- Procedimento:
    1. Para todo item da parte não-ordenada faça:
        - Procurar sua (nova) posição na parte já ordenada;

- Ordenação a partir de uma estrutura ordenada ou não-ordenada
- Separa a estrutura a ser ordenada em duas partes: já ordenada e não-ordenada
- Procedimento:
  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

- Ordenação a partir de uma estrutura ordenada ou não-ordenada
- Separa a estrutura a ser ordenada em duas partes: já ordenada e não-ordenada
- Procedimento:
  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
- Note que para vetores, a procura também envolve deslocar elementos para abrir espaço para inserção

- Considere Vetor A:

5	2	4	6	1	3

- Considere Vetor A:

5	2	4	6	1	3

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

- Considere Vetor A:

5	2	4	6	1	3

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

Desenvolvimento do exemplo no quadro

Iteração #0 $\Rightarrow A$:

5	2	4	6	1	3

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

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

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

Insertion Sort

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

- 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 $\Theta(n)$

- 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 $\Theta(n)$
  - ⋆ No pior caso, o tempo de execução é $\Theta(n^2)$

- 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 $\Theta(n)$
  - No pior caso, o tempo de execução é $\Theta(n^2)$
- É de fácil implementação

Ordenação por Inserção

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

- 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 $\Theta(n)$
  - No pior caso, o tempo de execução é $\Theta(n^2)$
- É de fácil implementação
- É relativamente eficiente para ordenar poucos dados

Ordenação por Inserção

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

- 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 $\Theta(n)$
  - No pior caso, o tempo de execução é $\Theta(n^2)$
- 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 $\Theta(n)$
  - No pior caso, o tempo de execução é $\Theta(n^2)$
- É de fácil implementação
- É relativamente eficiente para ordenar poucos dados
- É adaptativo, ou seja, eficiente para dados ordenados
- É estável, ou seja, 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 $\Theta(n)$
  - No pior caso, o tempo de execução é $\Theta(n^2)$
- É de fácil implementação
- É relativamente eficiente para ordenar poucos dados
- É adaptativo, ou seja, eficiente para dados ordenados
- É estável, ou seja, não altera a ordem relativa de elementos com chaves repetidas
- É *in-place*, ou seja, 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 $\Theta(n)$
  - No pior caso, o tempo de execução é $\Theta(n^2)$
- É de fácil implementação
- É relativamente eficiente para ordenar poucos dados
- É adaptativo, ou seja, eficiente para dados ordenados
- É estável, ou seja, não altera a ordem relativa de elementos com chaves repetidas
- É *in-place*, ou seja, apenas requer uma quantidade de memória constante $O(1)$ para funcionar
- É *online*, ou seja, pode ordenar uma lista à medida que a recebe

Ordenação por Seleção

- 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:
  - Vetor com muitos elementos

Ordenação por Seleção

- 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:
  - Vetor com muitos elementos
  - Vetor com elementos armazenados em dispositivos externos

Ordenação por Seleção

- 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:
  - 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 lista/vetor

- Em linhas gerais o algoritmo funciona da seguinte forma:
  1. Encontre o menor valor da 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 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 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, o vetor (ou lista) é dividido em duas partes:

- Em linhas gerais o algoritmo funciona da seguinte forma:

  1. Encontre o menor valor da 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, o vetor (ou lista) é dividido em duas partes:

  - a subvetor 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 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, o vetor (ou lista) é dividido em duas partes:
  - a subvetor de itens já ordenados, o qual é construído da esquerda para direita e localiza-se no início; e
  - a sublista de itens que precisam ser ordenados, ocupando o restante do vetor

Iteração #0 ⇒ A:

5	2	4	6	1	3

Iteração #1 ⇒ A:

5	2	4	6	1	3

Menor: $\boxed{1}$

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

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

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

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

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

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

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

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

Iteração #3 ⇒ A:

1	2	3	6	5	4

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

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

Ordenação por Seleção

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

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

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

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

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

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

Selection Sort

```
procedimento selectionSort(A: arranjo de ref inteiro)
    var n: inteiro ← tam A                                   # recuperar tamanho vetor
    var menor, i, j: inteiro                                 # variáveis auxiliares
    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
fim
```

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

- 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?*

- 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 serem ordenados estão dispostos em A

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

- O principal representante da filosofica 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 filosofica 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

Ordenação por Troca

(4) Algoritmo **Bubble Sort**

- O principal representante da filosofica 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 filosofica 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

Ordenação por Troca

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

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

Ordenação por Troca

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

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

	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
	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

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

Ordenação por Troca

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

Iteração # 3 $\Rightarrow A$:

2	4	1	3	5	6
2	4	1	3	5	6
2	1	4	3	5	6
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 ← tam A                          # recuperar tamanho vetor
    var i, j: inteiro                               # variáveis auxiliares
    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]) = −1 então
                A[i] ↔ A[i + 1]                     # realizar a troca
fim
```

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]) = −1 então
                A[i] ↔ A[i + 1]                         # realizar a troca
                houverTroca ← verdadeiro                # indicar troca
    até não houverTroca
fim
```

Ordenação por Troca

Bubble Sort (versão otimizada #2)

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

Ordenação por Troca

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

- 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 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 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 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, inlusive, 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, inlusive, possuem complexidade de pior caso $\Theta(n \lg n)$ também!
- Estes últimos são considerados **ótimos** para o problema de ordenação usando comparação de chaves

Referências

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.