# Projeto e Análise de Algoritmos
# Introdução

Antonio Luiz Basile

Faculdade de Computação e Informática
Universidade Presbiteriana Mackenzie

February 19, 2018

# Análise de Algoritmos

- Suponha que os computadores são infinitamente rápidos e de custo nenhum para memória.
- Será que ainda haveria algum motivo para estudar análise de algoritmos?
- A resposta é sim !
- Para demonstrar que sua solução termina e com a resposta certa.

# Eficiência

- Algoritmos distintos para resolver o mesmo problema frequentemente diferem muito em eficiência.
- Estas diferenças podem ser muito mais significantes que diferenças de software e hardware.
- Por exemplo, para o problema de ordenação podemos comparar o insertion-sort com o mergesort.
- O primeiro demora tempo proporcional a $n^2$.
- O segundo demora tempo proporcional a $n \log n$.

# Exemplo Prático

- Suponha o computador A mais rápido rodando Insertion Sort e computador B mais lento rodando Mergesort.
- Suponha que estejam rodando sobre um vetor de 10 milhões de elementos.
- Suponha que o computador A execute 10 bilhões de operações por segundo e que o computador B execute 10 milhões de operações por segundo. Portanto o computador A é 1000 vezes mais rápido que o computador B.
- Suponha que o Insertion Sort escrito em A é o mais otimizado possível e que requeira apenas $2n^2$ para ordenar $n$ elementos. Por outro lado, suponha que um péssimo Mergesort esteja rodando em B e que demore $50n \lg n$ instruções.
- O computador A demorará $5,5$ horas e o computador B menos de 20 minutos ($\sim$ 17 vezes mais rápido).

# Problema de ordenação: Formalismo

- **Entrada:** Uma sequência de $n$ números $< a_1, a_2, \cdots, a_n >$
- **Saída:** Uma permutação $< a_1^{'}, a_2^{'}, \cdots, a_n^{'} >$ da sequência de entrada tal que $< a_1^{'} \leq a_2^{'} \leq \cdots \leq a_n^{'} >$

# Insertion Sort (análogo ao jogo de cartas)

- Inicialmente não temos nenhuma carta na mão e todas as cartas viradas na mesa.
- A cada passo removemos uma carta por vez da mesa e a inserimos em sua posição correta entre as cartas que estão na mão.
- Para encontrar a posição correta de uma carta, comparamos esta carta com cada carta que já está na mão, da direita para a esquerda.
- Observe que durante todo o tempo as cartas que estão na mão estão ordenadas.

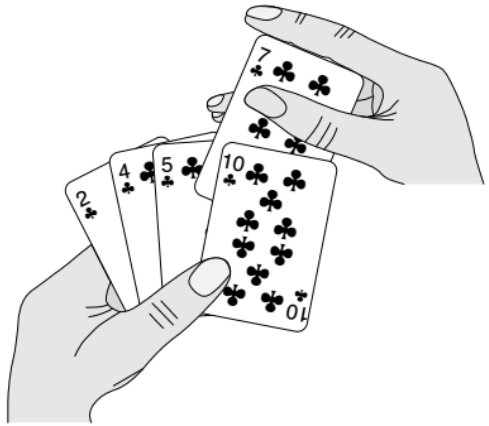


Figure: Jogo de cartas (from: CLR)

# Insertion Sort

```
ORDENAÇÃO-POR-INSERÇÃO (A, n)
1   para j crescendo de 2 até n faça
2       x ← A[j]
3       i ← j–1
4       enquanto i > 0 e A[i] > x faça
5           A[i+1] ← A[i]
6           i ← i–1
7       A[i+1] ← x
```

Figure: Insertion Sort (Feofiloff / CLR)

# Insertion Sort

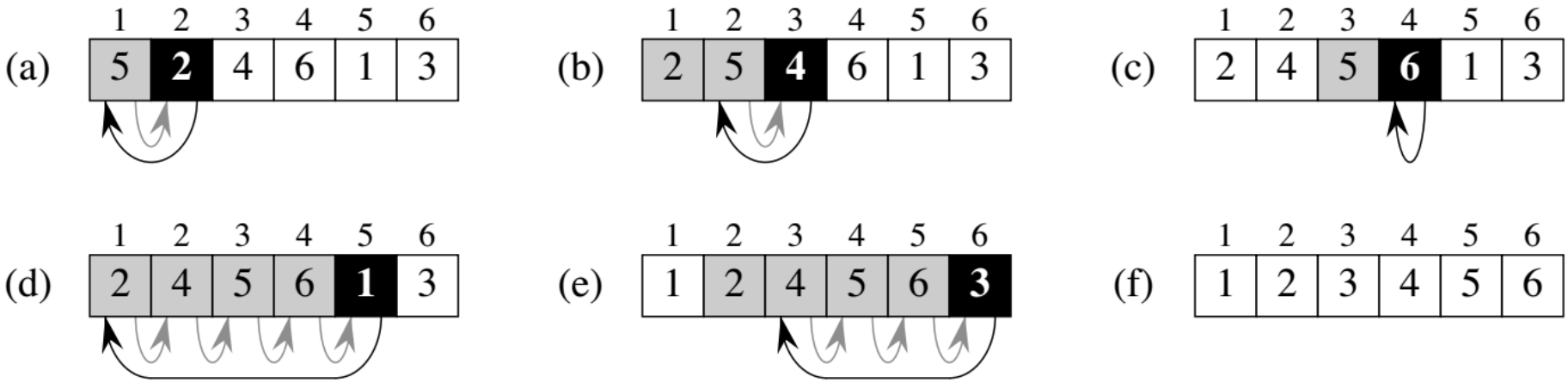



Figure: Insertion Sort (CLR)

# Correção

Para sabermos se um algoritmo é correto, ou seja, se faz o que promete, usamos **invariantes de laço** para nos ajudar. Precisamos mostrar 3 pontos sobre invariantes de laço:

1. **Inicialização:** É verdadeiro antes da primeira iteração do laço.
2. **Manutenção:** Se é verdadeiro antes de uma iteração do laço, permanece verdadeiro antes da próxima iteração.
3. **Término:** Quando o laço termina, o invariante nos dá uma propriedade útil que nos ajuda a mostrar que o algoritmo é correto.

Observe como o método acima é similar à indução matemática.

# Insertion Sort - Correção

1. **Inicialização:** Iniciamos mostrando que o invariante de laço vale antes da primeira iteração, quando $j = 2$. O subvetor $A[1 \cdots j-1]$, portanto, consiste de apenas um elemento A[1], trivialmente ordenado.
2. **Manutenção:** Em seguida mostramos que cada iteração mantém o invariante de laço. Informalmente, o corpo do laço **for** trabalha movendo $A[j-1], A[j-2], A[j-3]$, etc. uma posição para a direita até encontrar a posição correta para $A[j]$. O subvetor $A[i..j]$ então consiste dos elementos originais em $A[i..j]$, porém ordenados. Incrementando $j$ para o próximo laço **for** preserva, então, o invariante de laço.
3. **Término:** A condição que termina o laço **for** é $j = n + 1$. O subvetor $A[1..n]$ consiste dos elementos originais, mas ordenados. Como $A[1..n]$ é o vetor inteiro, concluímos que o vetor todo está ordenado. Logo, o algoritmo é correto.

# Insertion Sort - Análise

Analise o código abaixo.

```
ORDENAÇÃO-POR-INSERÇÃO (A, n)
1   para j crescendo de 2 até n faça
2       x ← A[j]
3       i ← j–1
4       enquanto i > 0 e A[i] > x faça
5           A[i+1] ← A[i]
6           i ← i–1
7       A[i+1] ← x
```

Quanto tempo o programa acima consome em número de operações?

# Insertion Sort - Análise

Em geral, o tempo que leva um algoritmo cresce com o tamanho da entrada, logo é comum descrever o tempo do programa em função do tamanho de sua entrada.

```
ORDENAÇÃO-POR-INSERÇÃO (A, n)
1  para j crescendo de 2 até n faça                n
2      x ← A[j]                                    n–1
3      i ← j–1                                     n–1
4      enquanto i > 0 e A[i] > x faça              2+3+...+n
5          A[i+1] ← A[i]                           1+2+3+...+n–1
6          i ← i–1                                 1+2+3+...+n–1
7      A[i+1] ← x                                  n–1
```

$$T(n) = (3/2)n^2 + (7/2)n - 4.$$

Figure: Insertion Sort (Feofiloff / CLR)

# Insertion Sort - Análise

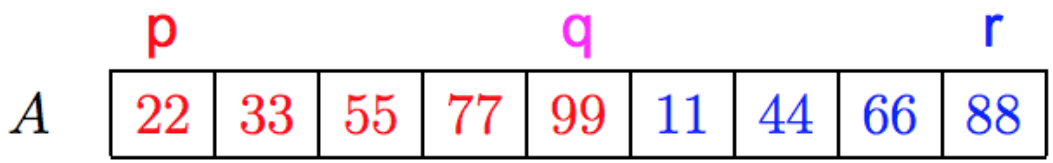
$$T(n) = (3/2)n^2 + (7/2)n - 4.$$

- A expressão de $T(n)$ é da forma $an^2 + bn + c$.
- O coeficiente $3/2$ de $n^2$ não é importante, pois não depende do algoritmo, mas de nossa hipótese *1 unidade de tempo por linha*.
- Já o $n^2$ é fundamental, pois caracteriza o algoritmo em si e não depende nem do computador nem dos detalhes de implementação do algoritmo.
- Dizemos que o algoritmo é quadrático.

# Intercalação

**Problema:** Dados $A[p..q]$ e $A[q+1..r]$ crescentes, rearranjar $A[p..r]$ de modo que ele fique em ordem crescente.

Entra:




Sai:

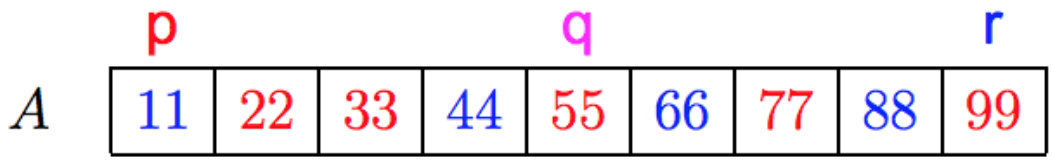



Figure: Intercala (Coelho)

# Intercalação

```
INTERCALA (A, p, q, r)
00   ▷ B[p..r] é um vetor auxiliar
01   para i ← p até q faça
02        B[i] ← A[i]
03   para j ← q + 1 até r faça
04        B[r + q + 1 − j] ← A[j]
05   i ← p
06   j ← r
07   para k ← p até r faça
08        se B[i] ≤ B[j]
09             então A[k] ← B[i]
10                  i ← i + 1
11             senão A[k] ← B[j]
12                  j ← j − 1
```

Figure: Intercala (Coelho)

# Intercalação

Se a execução de cada linha de código consome 1 unidade de tempo o consumo total será:

| linha | todas as execuções da linha |
|---|---|
| 1 | ? |
| 2 | ? |
| 3 | ? |
| 4 | ? |
| 5 | ? |
| 6 | ? |
| 7 | ? |
| 8 | ? |
| 9–12 | ? |
| total | ? |

Figure: Intercala (Coelho)