

## MC448 — Análise de Algoritmos I

Cid Carvalho de Souza   Cândida Nunes da Silva  
Orlando Lee

17 de agosto de 2009

## Agradecimentos (Cid e Cândida)

- Várias pessoas contribuíram **direta ou indiretamente** com a preparação deste material.
- Algumas destas pessoas cederam gentilmente seus arquivos digitais enquanto outras cederam gentilmente o seu tempo fazendo correções e dando sugestões.
- Uma lista destes “colaboradores” (**em ordem alfabética**) é dada abaixo:
  - Célia Picinin de Mello
  - José Coelho de Pina
  - Orlando Lee
  - Paulo Feofiloff
  - Pedro Rezende
  - Ricardo Dahab
  - Zaroni Dias

## Antes de mais nada. . .

- Uma versão anterior deste conjunto de slides foi preparada por Cid Carvalho de Souza e Cândida Nunes da Silva para uma instância anterior desta disciplina.
- O que vocês tem em mãos é uma versão modificada preparada para atender a meus gostos.
- Nunca é demais enfatizar que o material é apenas um **guia** e não deve ser usado como única fonte de estudo. Para isso consultem a bibliografia (em especial o CLR ou CLRS).

Orlando Lee

## Introdução

O nome da disciplina é **Análise de Algoritmos**.  
O que é então um algoritmo?

Informalmente, um **algoritmo** é um procedimento computacional bem definido que:

- recebe um conjunto de valores como **entrada** e
- produz um conjunto de valores como **saída**.

Um **algoritmo** é uma ferramenta para resolver um determinado **problema computacional**. A descrição do problema define a relação que deve existir entre a entrada e a saída do algoritmo.

## Exemplo: Problema da Ordenação

**Problema:** rearranjar um vetor  $A[1 \dots n]$  em ordem crescente.

**Entrada:**

1												$n$
33	55	33	44	33	22	11	99	22	55	77		

**Saída:**

1												$n$
11	22	22	33	33	33	44	55	55	77	99		

## MC448 - Análise de Algoritmos

O que veremos nesta disciplina?

- Como estimar a quantidade de **recursos** (**tempo**, **memória**) que um algoritmo consome/gasta = **análise de complexidade**
- Como provar a “**corretude**” de um algoritmo
- Como projetar algoritmos **eficientes** (= **rápidos**) para vários problemas computacionais

## Instância de um problema

Dizemos que o vetor

1												$n$
33	55	33	44	33	22	11	99	22	55	77		

é uma **instância** do problema de **ordenação**.

Em geral, uma **instância de um problema** é um conjunto de valores que serve de entrada para o problema (respeitando as restrições impostas na descrição deste).

## A importância dos algoritmos para a computação

O uso/desenvolvimento de algoritmos “eficientes” é desejável em vários contextos:

- projetos de genoma de seres vivos
- rede mundial de computadores
- sistemas de informação geográfica
- comércio eletrônico
- planejamento da produção de indústrias
- logística de distribuição
- ...

## Algoritmos e tecnologia

- O avanço da tecnologia permite a construção de máquinas cada vez mais rápidas. Isto possibilita que um algoritmo para determinado problema possa ser executado mais **rapidamente**.
- Paralelamente a isto, há o projeto/desenvolvimento de algoritmos “**intrinsecamente mais eficientes**” para determinado problema. Isto leva em conta apenas as características inerentes ao problema, desconsiderando detalhes de software/hardware.
- Vamos “comparar” estes dois aspectos através de um exemplo.

## Algoritmos e tecnologia

### Exemplo: ordenação de um vetor de $n$ elementos

- Suponha que os computadores  $A$  e  $B$  executam  $1G$  e  $10M$  instruções por segundo, respectivamente. Ou seja,  **$A$  é 100 vezes mais rápido que  $B$** .
- **Algoritmo 1**: implementado em  $A$  por um excelente programador em linguagem de máquina (ultra-rápida). Executa  $2n^2$  instruções.
- **Algoritmo 2**: implementado na máquina  $B$  por um programador mediano em linguagem de alto nível dispondo de um compilador “meia-boca”. Executa  $50n \log_{10} n$  instruções.

## Algoritmos e tecnologia

- O que acontece quando ordenamos um vetor de **um milhão de elementos**? **Qual algoritmo é mais rápido?**
- **Algoritmo 1 na máquina  $A$** :  
$$\frac{2 \cdot (10^6)^2 \text{ instruções}}{10^9 \text{ instruções/segundo}} \approx 2000 \text{ segundos}$$
- **Algoritmo 2 na máquina  $B$** :  
$$\frac{50 \cdot (10^6 \log 10^6) \text{ instruções}}{10^7 \text{ instruções/segundo}} \approx 100 \text{ segundos}$$
- Ou seja,  **$B$  foi VINTE VEZES mais rápido do que  $A$ !**
- Se o vetor tiver **10 milhões de elementos**, esta razão será de **2.3 dias** para **20 minutos!**

## Algoritmos e tecnologia

- O uso de um **algoritmo** em lugar de outro pode levar a ganhos extraordinários de **desempenho**.
- Isso pode ser tão importante quanto o projeto de **hardware**.
- A melhoria obtida pode ser tão significativa que não poderia ser obtida simplesmente com o avanço da tecnologia.

## Complexidade de algoritmos

- Queremos projetar/desenvolver **algoritmos eficientes** (**rápidos**).
- Mas o que seria uma boa **medida de eficiência** de um algoritmo?
- Não estamos interessados em quem programou, em que linguagem foi escrito e nem qual a máquina foi usada!
- Queremos um critério uniforme para **comparar algoritmos**.

## Inserção em um vetor ordenado

1						$j$				$n$
20	25	35	40	44	55	38	99	10	65	50

- O subvetor  $A[1 \dots j - 1]$  está **ordenado**.
- Queremos inserir a **chave = 38 =  $A[j]$**  em  $A[1 \dots j - 1]$  de modo que no final tenhamos:

1						$j$				$n$
20	25	35	38	40	44	55	99	10	65	50

- Agora  $A[1 \dots j]$  está ordenado.

## Exemplo: Problema da Ordenação

**Problema:** ordenar um vetor em ordem crescente

**Entrada:** um vetor  $A[1 \dots n]$

**Saída:** vetor  $A[1 \dots n]$  rearranjado em ordem crescente

Vamos começar analisando o algoritmo de ordenação baseado no **método de inserção** (**Insertion sort**).

Isto nos permitirá destacar alguns dos aspectos mais importantes no estudo de algoritmos para esta disciplina.

## Como fazer a inserção

1						$i$	$j$			$n$
20	25	35	40	44	55	38	99	10	65	50

1					$i$		$j$			$n$
20	25	35	40	44		55	99	10	65	50

1					$i$		$j$			$n$
20	25	35	40		44	55	99	10	65	50

1					$i$		$j$			$n$
20	25	35		40	44	55	99	10	65	50

1					$i$		$j$			$n$
20	25	35	38	40	44	55	99	10	65	50

## Ordenação por inserção

<i>chave</i>	1							<i>j</i>			<i>n</i>
99	20	25	35	38	40	44	55	99	10	65	50

<i>chave</i>	1							<i>j</i>			<i>n</i>
99	20	25	35	38	40	44	55	99	10	65	50

<i>chave</i>	1							<i>j</i>			<i>n</i>
10	20	25	35	38	40	44	55	99	10	65	50

<i>chave</i>	1							<i>j</i>			<i>n</i>
10	10	20	25	35	38	40	44	55	99	65	50

## Descrição um algoritmo

Podemos formalizar o algoritmo ORDENA-POR-INSERÇÃO de várias formas:

- usando uma linguagem de programação de alto nível: C, Pascal, Java etc
- implementando-o em linguagem de máquina diretamente executável em *hardware*
- em português
- em um pseudo-código de alto nível, como no livro do CLRS

Usaremos essencialmente as duas últimas alternativas nesta disciplina.

## Ordenação por inserção

<i>chave</i>	1								<i>j</i>		<i>n</i>
65	10	20	25	35	38	40	44	55	99	65	50

<i>chave</i>	1								<i>j</i>		<i>n</i>
65	10	20	25	35	38	40	44	55	65	99	50

<i>chave</i>	1									<i>j</i>	
50	10	20	25	35	38	40	44	55	65	99	50

<i>chave</i>	1										<i>j</i>
50	10	20	25	35	38	40	44	50	55	65	99

## Ordena-Por-Inserção

### Pseudo-código

```
ORDENA-POR-INSERÇÃO(A, n)
1  para j ← 2 até n faça
2    chave ← A[j]
3    ▷ Insere A[j] no subvetor ordenado A[1..j - 1]
4    i ← j - 1
5    enquanto i ≥ 1 e A[i] > chave faça
6      A[i + 1] ← A[i]
7      i ← i - 1
8    A[i + 1] ← chave
```

## Análise do algoritmo

O que é importante analisar/considerar?

- **Corretude do algoritmo:** é preciso mostrar que para toda instância do problema, o algoritmo **pára** e devolve uma **resposta correta**.
- **Complexidade de tempo do algoritmo:** quantas instruções são necessárias no pior caso para ordenar os  $n$  elementos?

## O algoritmo pára

---

ORDENA-POR-INserÇÃO( $A, n$ )

```
1  para  $j \leftarrow 2$  até  $n$  faça
    ...
4       $i \leftarrow j - 1$ 
5      enquanto  $i \geq 1$  e  $A[i] > chave$  faça
6          ...
7           $i \leftarrow i - 1$ 
8      ...
```

---

No **laço enquanto** na linha 5 o valor de  $i$  **diminui a cada iteração** e o **valor inicial** é  $i = j - 1 \geq 1$ . Logo, a sua execução pára em algum momento por causa do teste condicional  $i \geq 1$ .

O **laço na linha 1** evidentemente **pára** (o contador  $j$  atingirá o valor  $n + 1$  após  $n - 1$  iterações).

Portanto, o algoritmo **pára**.

## Ordena-Por-Inserção

---

ORDENA-POR-INserÇÃO( $A, n$ )

```
1  para  $j \leftarrow 2$  até  $n$  faça
2      chave  $\leftarrow A[j]$ 
3       $\triangleright$  Insere  $A[j]$  no subvetor ordenado  $A[1..j - 1]$ 
4       $i \leftarrow j - 1$ 
5      enquanto  $i \geq 1$  e  $A[i] > chave$  faça
6           $A[i + 1] \leftarrow A[i]$ 
7           $i \leftarrow i - 1$ 
8       $A[i + 1] \leftarrow chave$ 
```

---

O que falta fazer?

- Verificar se ele produz uma **resposta correta**.
- Analisar sua **complexidade de tempo**.

## Invariantes de laço e provas de corretude

- **Definição:** um **invariante de um laço** é uma **propriedade** que relaciona as variáveis do algoritmo a cada execução completa do laço.
- Ele deve ser escolhido de modo que, ao término do laço, tenha-se uma propriedade útil para mostrar a corretude do algoritmo.
- A prova de corretude de um algoritmo requer que sejam encontrados e provados invariantes dos vários laços que o compõem.
- Em geral, é **mais difícil** descobrir um **invariante apropriado** do que mostrar sua validade se ele for dado de bandeja. . .

## Exemplo de invariante

ORDENA-POR-INSERÇÃO( $A, n$ )

```
1 para  $j \leftarrow 2$  até  $n$  faça
2    $chave \leftarrow A[j]$ 
3   ▷ Insere  $A[j]$  no subvetor ordenado  $A[1..j-1]$ 
4    $i \leftarrow j-1$ 
5   enquanto  $i \geq 1$  e  $A[i] > chave$  faça
6      $A[i+1] \leftarrow A[i]$ 
7      $i \leftarrow i-1$ 
8    $A[i+1] \leftarrow chave$ 
```

### Invariante principal de ORDENA-POR-INSERÇÃO: (i1)

No começo de cada iteração do laço **para** das linha 1–8, o subvetor  $A[1 \dots j-1]$  está ordenado.

## Corretude de algoritmos por invariantes

A estratégia “típica” para mostrar a corretude de um algoritmo iterativo através de invariantes segue os seguintes passos:

- 1 Mostre que o invariante **vale** no início da **primeira iteração** (trivial, em geral)
- 2 Suponha que o invariante **vale** no início de uma **iteração qualquer** e prove que ele **vale** no início da **próxima iteração**
- 3 Conclua que se o algoritmo **pára** e o invariante **vale** no início da **última iteração**, então o algoritmo é **correto**.

Note que (1) e (2) implicam que o invariante vale no início de qualquer iteração do algoritmo. Isto é similar ao método de **indução matemática** ou **indução finita**!

## Corretude da ordenação por inserção

Vamos verificar a **corretude do algoritmo de ordenação por inserção** usando a técnica de **prova por invariantes de laços**.

### Invariante principal: (i1)

No começo de cada iteração do laço **para** das linhas 1–8, o subvetor  $A[1 \dots j-1]$  está ordenado.

1						$j$				$n$
20	25	35	40	44	55	38	99	10	65	50

- Suponha que o invariante vale.
- Então a corretude do algoritmo é “evidente”. **Por quê?**
- No início da última iteração temos  $j = n + 1$ . Assim, do invariante segue que o (sub)vetor  $A[1 \dots n]$  está ordenado!

## Melhorando a argumentação

ORDENA-POR-INSERÇÃO( $A, n$ )

```
1 para  $j \leftarrow 2$  até  $n$  faça
2    $chave \leftarrow A[j]$ 
3   ▷ Insere  $A[j]$  no subvetor ordenado  $A[1 \dots j-1]$ 
4    $i \leftarrow j-1$ 
5   enquanto  $i \geq 1$  e  $A[i] > chave$  faça
6      $A[i+1] \leftarrow A[i]$ 
7      $i \leftarrow i-1$ 
8    $A[i+1] \leftarrow chave$ 
```

### Um invariante mais preciso: (i1')

No começo de cada iteração do laço **para** das linhas 1–8, o subvetor  $A[1 \dots j-1]$  é uma permutação ordenada do subvetor original  $A[1 \dots j-1]$ .

## Esboço da demonstração de (i1')

- 1 Validade na primeira iteração: neste caso, temos  $j = 2$  e o invariante simplesmente afirma que  $A[1 \dots 1]$  está ordenado, o que é evidente.
- 2 Validade de uma iteração para a seguinte: segue da discussão anterior. O algoritmo **empurra** os elementos maiores que a **chave** para seus lugares corretos e ela é colocada no **espaço vazio**.  
**Uma demonstração mais formal deste fato exige invariantes auxiliares para o laço interno enquanto.**
- 3 Corretude do algoritmo: na última iteração, temos  $j = n + 1$  e logo  $A[1 \dots n]$  está ordenado com os **elementos originais** do vetor. Portanto, o algoritmo é **correto**.

## Complexidade do algoritmo

- Vamos tentar determinar o **tempo de execução** (ou **complexidade de tempo**) de ORDENA-POR-INSERÇÃO em função do **tamanho de entrada**.
- Para o **Problema de Ordenação** definimos como tamanho de entrada a **número de elementos do vetor**.
- A **complexidade de tempo** de um algoritmo é o número de **instruções básicas** (operações elementares ou primitivas) que executa a partir de uma entrada.
- **Exemplo:** comparação e atribuição entre números ou variáveis numéricas, operações aritméticas, etc.

## Invariantes auxiliares

No início da linha 5 valem os seguintes invariantes:

- (i2)  $A[1 \dots i]$  e  $A[i + 2 \dots j]$  contêm os elementos de  $A[1 \dots j]$  antes de entrar no laço que começa na linha 5.
- (i3)  $A[1 \dots i]$  e  $A[i + 2 \dots j]$  são crescentes.
- (i4)  $A[1 \dots i] \leq A[i + 2 \dots j]$
- (i5)  $A[i + 2 \dots j] > \text{chave}$ .

Invariantes (i2) a (i5)  
+ condição de parada na linha 5  
+ atribuição da linha 7 }  $\implies$  invariante (i1')

**Demonstração? Mesma que antes.**

## Vamos contar ?

ORDENA-POR-INSERÇÃO( $A, n$ )	Custo	# execuções
1 <b>para</b> $j \leftarrow 2$ <b>até</b> $n$ <b>faça</b>	$c_1$	?
2 <b>chave</b> $\leftarrow A[j]$	$c_2$	?
3 $\triangleright$ Insere $A[j]$ em $A[1 \dots j - 1]$	0	?
4 $i \leftarrow j - 1$	$c_4$	?
5 <b>enquanto</b> $i \geq 1$ e $A[i] > \text{chave}$ <b>faça</b>	$c_5$	?
6 $A[i + 1] \leftarrow A[i]$	$c_6$	?
7 $i \leftarrow i - 1$	$c_7$	?
8 $A[i + 1] \leftarrow \text{chave}$	$c_8$	?

O valor  $c_k$  representa o **custo (tempo)** de cada execução da linha  $k$ .

Denote por  $t_j$  o **número de vezes** que o teste no laço **enquanto** na linha 5 é feito para aquele valor de  $j$ .



## Vamos contar ?

ORDENA-POR-INSERÇÃO( $A, n$ )	Custo	Vezes
1 <b>para</b> $j \leftarrow 2$ <b>até</b> $n$ <b>faça</b>	$c_1$	$n$
2 <b>chave</b> $\leftarrow A[j]$	$c_2$	$n - 1$
3 $\triangleright$ Insere $A[j]$ em $A[1 \dots j - 1]$	0	$n - 1$
4 $i \leftarrow j - 1$	$c_4$	$n - 1$
5 <b>enquanto</b> $i \geq 1$ e $A[i] >$ <b>chave</b> <b>faça</b>	$c_5$	$\sum_{j=2}^n t_j$
6 $A[i + 1] \leftarrow A[i]$	$c_6$	$\sum_{j=2}^n (t_j - 1)$
7 $i \leftarrow i - 1$	$c_7$	$\sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] \leftarrow$ <b>chave</b>	$c_8$	$n - 1$

O valor  $c_k$  representa o **custo (tempo)** de cada execução da linha  $k$ .

Denote por  $t_j$  o **número de vezes** que o teste no laço **enquanto** na linha 5 é feito para aquele valor de  $j$ .

## Melhor caso

O **melhor caso** de Ordena-Por-Inserção ocorre quando o vetor  $A$  já está **ordenado**. Para  $j = 2, \dots, n$  temos  $A[j] \leq$  **chave** na linha 5 quando  $i = j - 1$ . Assim,  $t_j = 1$  para  $j = 2, \dots, n$ .

Logo,

$$\begin{aligned} T(n) &= c_1 n + c_2(n - 1) + c_4(n - 1) + c_5(n - 1) + c_8(n - 1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Este tempo de execução é da forma  $an + b$  para constantes  $a$  e  $b$  que dependem apenas dos  $c_i$ .

Portanto, **no melhor caso**, o tempo de execução é uma **função linear** no **tamanho da entrada**.

## Tempo de execução total

Logo, o tempo total de execução  $T(n)$  de Ordena-Por-Inserção é a soma dos tempos de execução de cada uma das linhas do algoritmo, ou seja:

$$\begin{aligned} T(n) &= c_1 n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=2}^n t_j \\ &\quad + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) \\ &\quad + c_8(n - 1) \end{aligned}$$

Como se vê, entradas de **tamanho igual** (i.e., mesmo valor de  $n$ ), podem apresentar **tempos de execução diferentes** já que o valor de  $T(n)$  depende dos valores dos  $t_j$ .

## Pior Caso

Quando o vetor  $A$  está em **ordem decrescente**, ocorre o **pior caso** para Ordena-Por-Inserção. Para inserir a **chave** em  $A[1 \dots j - 1]$ , temos que compará-la com todos os elementos neste subvetor. Assim,  $t_j = j$  para  $j = 2, \dots, n$ .

Lembre-se que:

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1$$

e

$$\sum_{j=2}^n (j - 1) = \frac{n(n-1)}{2}.$$

## Pior caso – continuação

Temos então que

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left( \frac{n(n+1)}{2} - 1 \right) \\ &\quad + c_6 \left( \frac{n(n-1)}{2} \right) + c_7 \left( \frac{n(n-1)}{2} \right) + c_8(n-1) \\ &= \left( \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left( c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

O tempo de execução no pior caso é da forma  $an^2 + bn + c$  onde  $a, b, c$  são constantes que dependem apenas dos  $c_i$ .

Portanto, **no pior caso**, o tempo de execução é uma **função quadrática** no **tamanho da entrada**.

## Complexidade assintótica de algoritmos

- Na maior parte desta disciplina, consideraremos a **análise de pior caso** e o **comportamento assintótico** de um algoritmo (instâncias de **tamanho grande**).
- O algoritmo **ORDENA-POR-INSERÇÃO** tem como complexidade (de **pior caso**) uma função quadrática  $an^2 + bn + c$ , onde  $a, b, c$  são constantes absolutas que dependem apenas dos custos  $c_i$ .
- O estudo assintótico nos permite “jogar para debaixo do tapete” os valores destas constantes, i.e., aquilo que independe do tamanho da entrada (neste caso os valores de  $a, b$  e  $c$ ).
- **Por que podemos fazer isso ?**

## Análise assintótica de funções quadráticas

Considere a função quadrática  $3n^2 + 10n + 50$ :

$n$	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

Como se vê,  $3n^2$  é o termo dominante quando  $n$  é grande.

De um modo geral, **podemos nos concentrar nos termos dominantes** e esquecer os demais.

## Notação assintótica

- Usando notação assintótica, dizemos que o algoritmo Ordena-Por-Inserção tem **complexidade de tempo de pior caso**  $\Theta(n^2)$ .
- Isto quer dizer **duas** coisas:
  - a complexidade de tempo é limitada (**superiormente**) assintoticamente por algum polinômio da forma  $an^2$  para alguma constante  $a$ ,
  - para todo  $n$  suficientemente grande, existe alguma instância de tamanho  $n$  que consome tempo **por pelo menos**  $dn^2$ , para alguma contante positiva  $d$ .
- **Mais adiante no curso discutiremos em detalhes o uso da notação assintótica em análise de algoritmos.**

## Tamanho da entrada

- O tempo gasto por um **algoritmo de ordenação** depende da entrada: ordenar 1000 números é mais demorado do que ordenar 3 números.
- Em geral, o tempo gasto por um algoritmo é maior quanto maior for o **tamanho da entrada**. A ideia então é **medir/estimar o tempo** gasto pelo algoritmo **em função do tamanho da entrada**.
- A noção exata de **tamanho da entrada** depende do problema em consideração. No caso do **Problema de Ordenação** é simplesmente o **número de elementos** da seqüência de entrada.

## Modelo Computacional

Quando analisamos o algoritmo ORDENA-POR-INSERÇÃO, implicitamente fizemos algumas hipóteses sobre o funcionamento do algoritmo:

- uma comparação leva tempo constante
- uma operação aritmética leva tempo constante
- permite acesso direto à memória
- controle de fluxo de laços/teste leva tempo constante

Formalmente, o que fizemos foi analisar o comportamento do algoritmo dentro de um **modelo computacional**.

## Modelo Computacional

- O modelo computacional estabelece quais os recursos disponíveis, as **instruções básicas** e quanto elas custam (= **tempo**).
- Dentre desse modelo, tentamos estimar através de uma **análise matemática** o tempo que um algoritmo gasta em função do **tamanho da entrada** (= **análise de complexidade**).
- A análise de complexidade depende **sempre** do modelo computacional adotado.

## Máquinas RAM

Salvo mencionado o contrário, usaremos o **Modelo Abstrato RAM** (Random Access Machine):

- o modelo RAM “simula” uma máquina de verdade
- permite acesso direto à memória
- possui um único processador que executa instruções **seqüencialmente**
- tipos básicos são números inteiros e reais
- há um limite no tamanho de cada *palavra de memória*: se a entrada tem “tamanho”  $n$ , então cada inteiro/real é representado por  $c \log n$  bits onde  $c \geq 1$  é uma constante

## Máquinas RAM

- executa **operações aritméticas** (soma, subtração, multiplicação, divisão, piso, teto), **comparações**, **movimentação de dados** de tipo básico e **fluxo de controle** (teste *if/else*, chamada e retorno de rotinas) em **tempo constante**,
- o tempo de execução de certas operações caem em uma **zona cinza**, por exemplo, **exponenciação**,
- veja maiores detalhes do modelo RAM no CLRS.

## Medida de eficiência de algoritmos

- Um algoritmo é chamado **eficiente** se a função que mede sua **complexidade de tempo** é **limitada** por um **polinômio** no tamanho da entrada.  
Por exemplo:  $n$ ,  $3n - 7$ ,  $n \log n$ ,  $4n^2$ ,  $143n^2 - 4n + 2$ ,  $n^5$ .
- Mas por que **polinômios**?  
Resposta padrão: (**polinômios são funções bem "comportadas"**).

## Medida de complexidade de algoritmos

- A **complexidade de tempo** (= **eficiência**) de um algoritmo é o **número de instruções básicas** que ele executa em **função do tamanho da entrada**.
- Adotamos uma "atitude pessimista" e em geral fazemos uma **análise de pior caso**.  
Determinamos o **tempo máximo necessário** para resolver uma instância de um certo **tamanho**.
- Além disso, a análise concentra-se no comportamento do algoritmo para entradas de tamanho **GRANDE** = **análise assintótica**.

## Vantagens do método de análise proposto

- O modelo RAM é robusto e permite **prever** o comportamento de um algoritmo para instâncias **GRANDES**.
- O modelo permite **comparar** algoritmos que resolvem um mesmo problema.
- A análise é mais robustas em relação às evoluções tecnológicas .

## Desvantagens do método de análise proposto

- Fornece um limite de **complexidade** pessimista sempre considerando o **pior caso**.
- Em uma aplicação real, nem todas as instâncias ocorrem com a mesma frequência e é possível que as "**instâncias ruins**" ocorram raramente.
- Não fornece nenhuma informação sobre o comportamento do algoritmo no **caso médio**.
- A análise de **complexidade de algoritmos** no **caso médio** é bastante **difícil**, principalmente, porque muitas vezes não é claro o que é o "**caso médio**".