

Projeto e Análise de Algoritmos

## Aula 2:

# Função de Complexidade Notação Assintótica

DECOM/UFOP

Anderson Almeida Ferreira

Adaptado do material feito por Andréa Iabrudi  
Tavares



# Qual o modelo para medir desempenho?

- Medida precisa de um programa executando em um ambiente específico?
  - muitas variáveis externas
- Algoritmo em pseudo-código para RAM
  - Capturar os aspectos essenciais
    - Abstração de hardware, sistema operacional, compilador, linguagem, etc
  - Medir número de passos (operações básicas)
  - **Custo constante** para passo

# Função de Complexidade

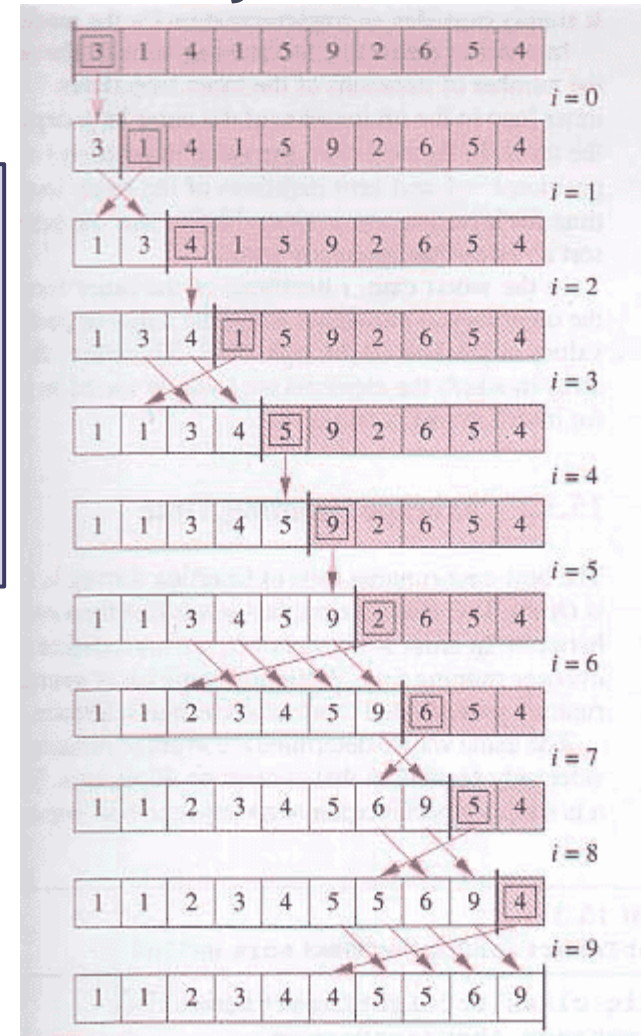
$$T(n)$$

- Número de passos/itens de memória
- Depende do tamanho da entrada
  - Número de elementos
  - Número de bits
  - Número de arestas + vértices
- Depende da entrada: melhor/pior caso/ caso médio

# Exemplo: Ordenação por Inserção

function insertion(A[n])

1. for  $i=2..n$ :
2.    $elem = A[i]$
3.    $j = i$
4.   while  $(j>1)$  and  $(elem < A[j-1])$ :
5.      $A[j] = A[j-1]$
6.      $j = j - 1$
7.    $A[j] = elem$



# Inserção - Função de complexidade

```

function insertion(A[n])
1. for i=2..n:
2.   elem = A[i]
3.   j = i
4.   while (j>1) and (elem<A[j-1]):
5.     A[j] = A[j-1]
6.     j = j-1
7.   A[j] = elem
  
```

	# instruções	vezes
1	2	n
2	1	(n-1)
3	1	(n-1)
4	3	(ti+1) p/ i
5	1	ti p/i
6	1	ti p/i
7	1	(n-1)

**For**  $\Rightarrow$ 

Atribuição	Teste
Incremento	Teste
Incremento	Teste
Incremento	Teste
...	...
<b>Incremento</b>	<b>Teste (F)</b>

$$T(n) = 2n + 3(n-1) + \sum_{i=2}^n 3(ti + 1) + 2\sum_{i=2}^n ti$$

$$T(n) = 5n - 3 + 5\sum_{i=2}^n ti + 3\sum_{i=2}^n 1$$

$$T(n) = 5n - 3 + 3(n-1) + 5\sum_{i=2}^n ti$$

$$T(n) = 8n - 6 + 5\sum_{i=2}^n ti$$

# Inserção - Função de complexidade

$$T(n) = 8n - 6 + 5\sum_{i=2}^n ti$$

- Melhor caso

- $ti = 0$
- $T(n) = 8n - 6$

- Pior caso

- $ti = i-1$
- $T(n) = 8n - 6 + 5\sum_{i=2}^n (i-1)$
- $T(n) = 8n - 6 + 5\sum_{i=1}^{n-1} i$
- $T(n) = 8n - 6 + 5n(n-1)/2$
- $T(n) = (5n^2 + 11n - 12)/2$

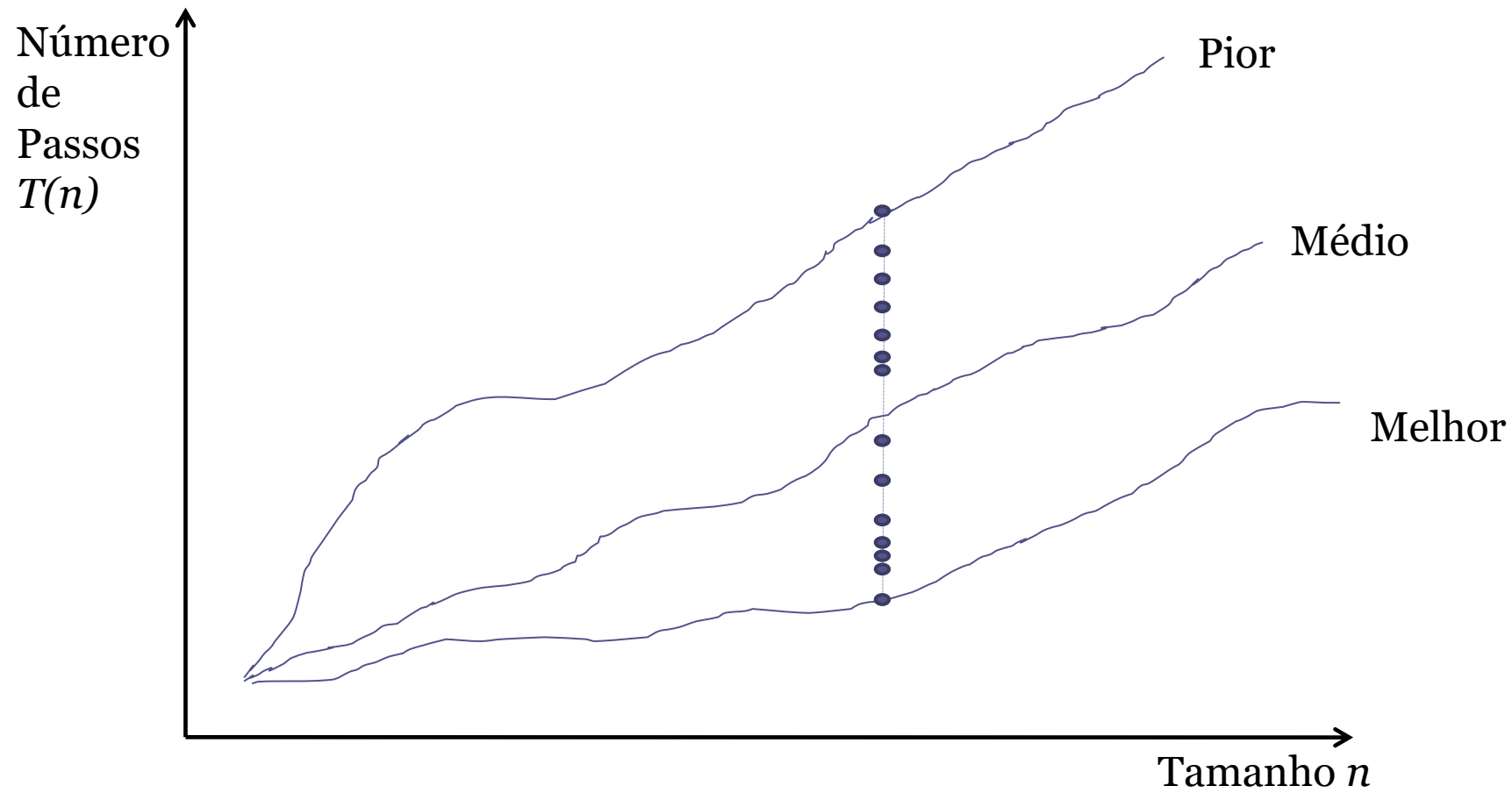
- Caso médio

- Assumindo probabilidades iguais
- $ti \text{ médio} = \frac{1}{i} (0 + 1 + \dots + (i-1))$
- $= \frac{1}{i} \sum_{j=0}^{i-1} j$
- $= \frac{1}{i} i(i-1)/2$
- $ti \text{ médio} = (i-1)/2$
- $T(n) = 8n - 6 + 5\sum_{i=2}^n (i-1)/2$
- $T(n) = 8n - 6 + \frac{5}{2} \sum_{i=1}^{n-1} i$
- $T(n) = 8n - 6 + 5n(n-1)/4$
- $T(n) = (5n^2 + 27n - 24)/4$

# Função de Complexidade

- Caso médio: tempo esperado
- Melhor caso: menor tempo de execução
- Pior caso : maior tempo de execução

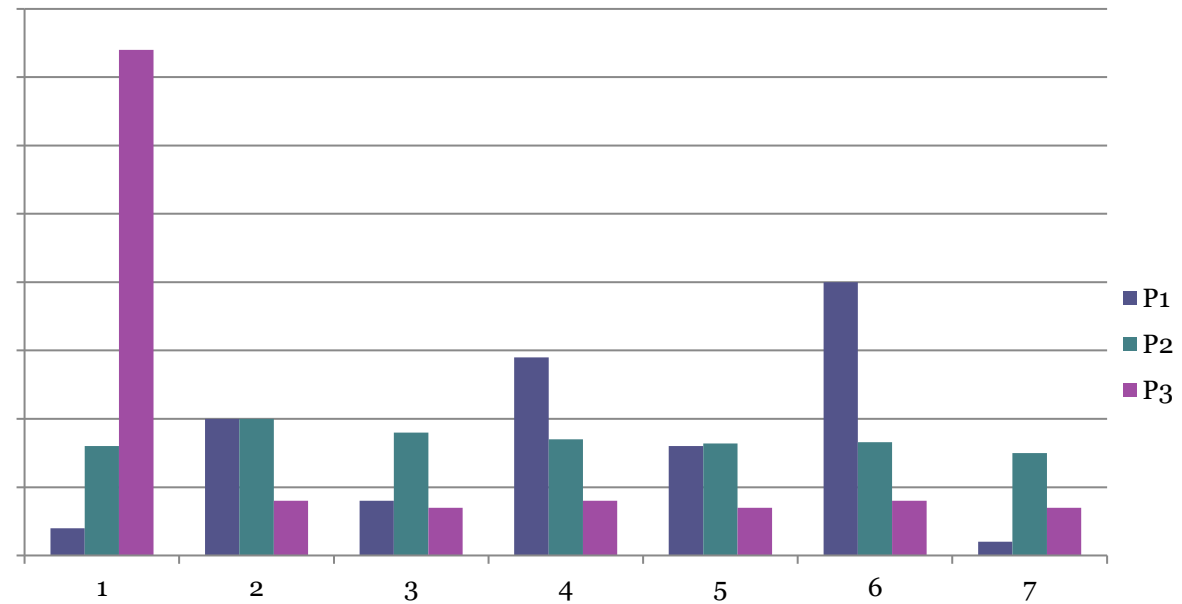
# Funções de Complexidade





# Qual o melhor programa?

D1	20	80	370
D2	100	100	40
D3	40	90	35
D4	145	85	40
D5	80	82	35
D6	200	83	40
D7	10	75	35
Médio	85	85	85
Pior	200	100	370
Melhor	10	75	35



# Diferentes medidas

- Caso médio: tempo esperado
  - Não se sabe a distribuição da entrada
  - Pode ser difícil de calcular
- Melhor caso: menor tempo de execução
  - Normalmente, pouquíssimas instâncias
  - Pode enganar
- Pior caso : maior tempo de execução
  - garantia de tempo de execução

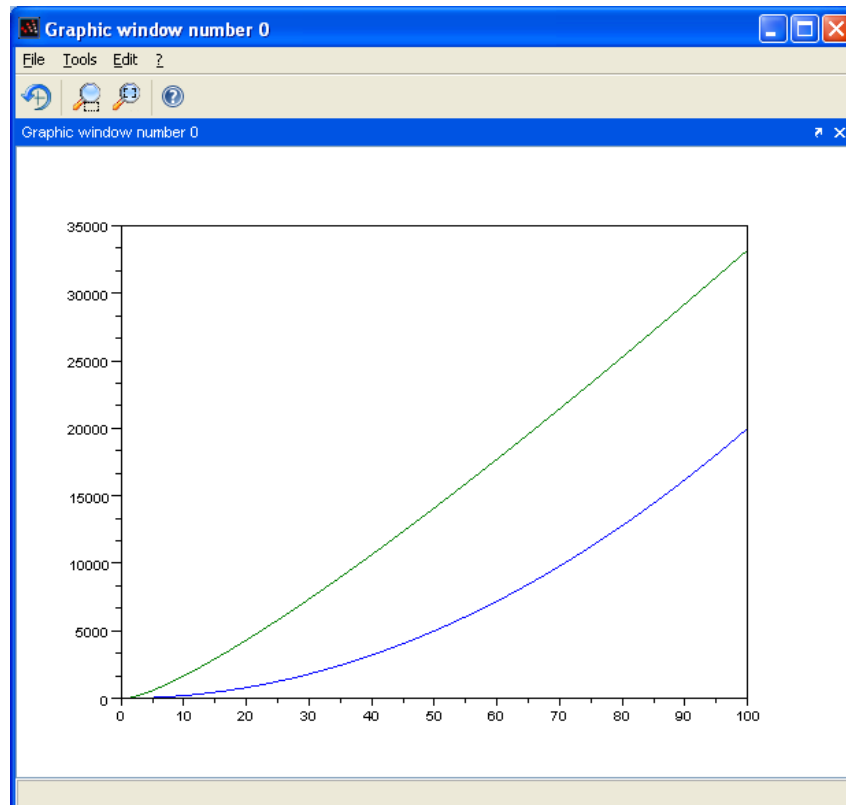
# Comparação Efetiva

- Inserção com melhor programador:
  - $T_1(n) = 2n^2$
- MergeSort com programador medíocre:
  - $T_2(n) = 50 n \lg n$
- No mesmo computador, qual é o melhor se a entrada for 100? E se for 2000?

<b>n</b>	<b><math>2 \cdot n^2</math></b>	<b><math>50n \lg n</math></b>
100	20.000	33.219,28
2000	8.000.000	1.096.578,43

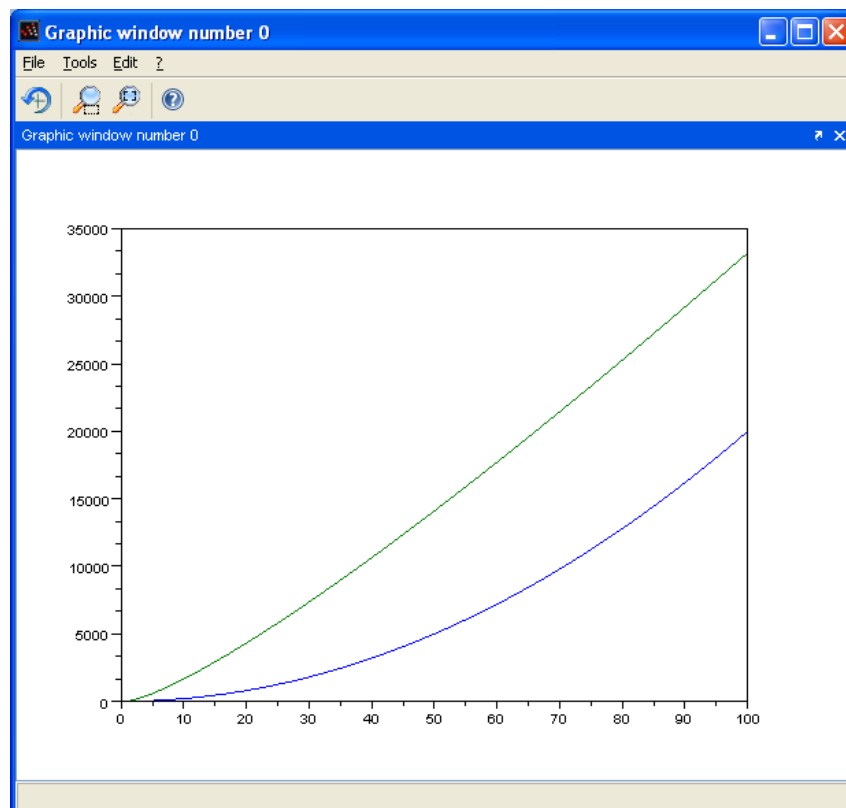
# Execução num mesmo computador

Entradas Pequenas

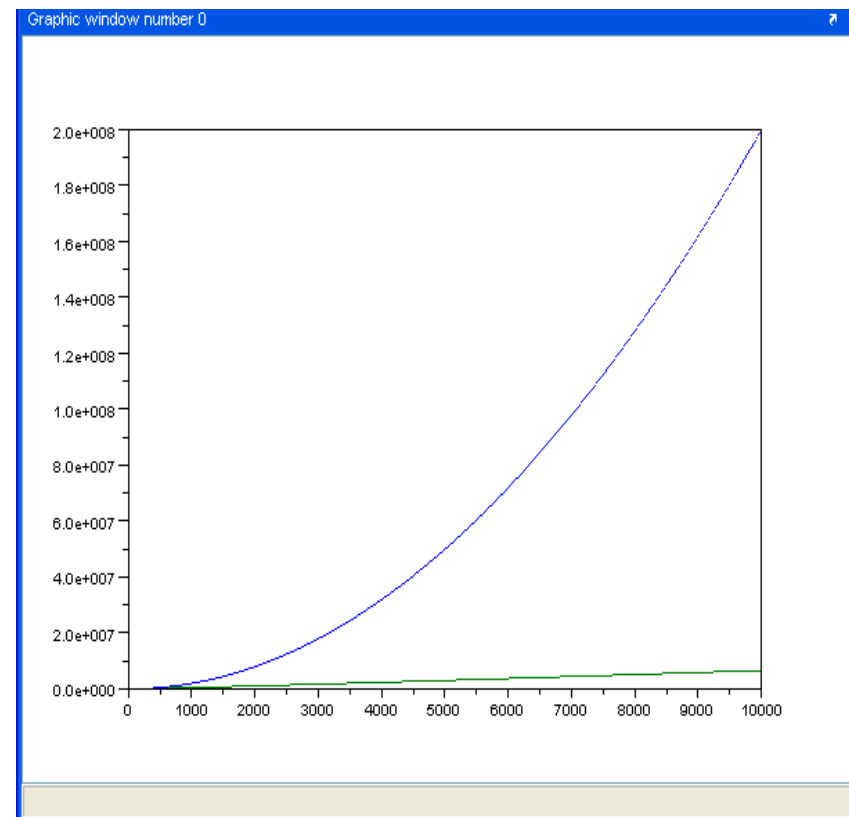


# Execução num mesmo computador

## Entradas Pequenas

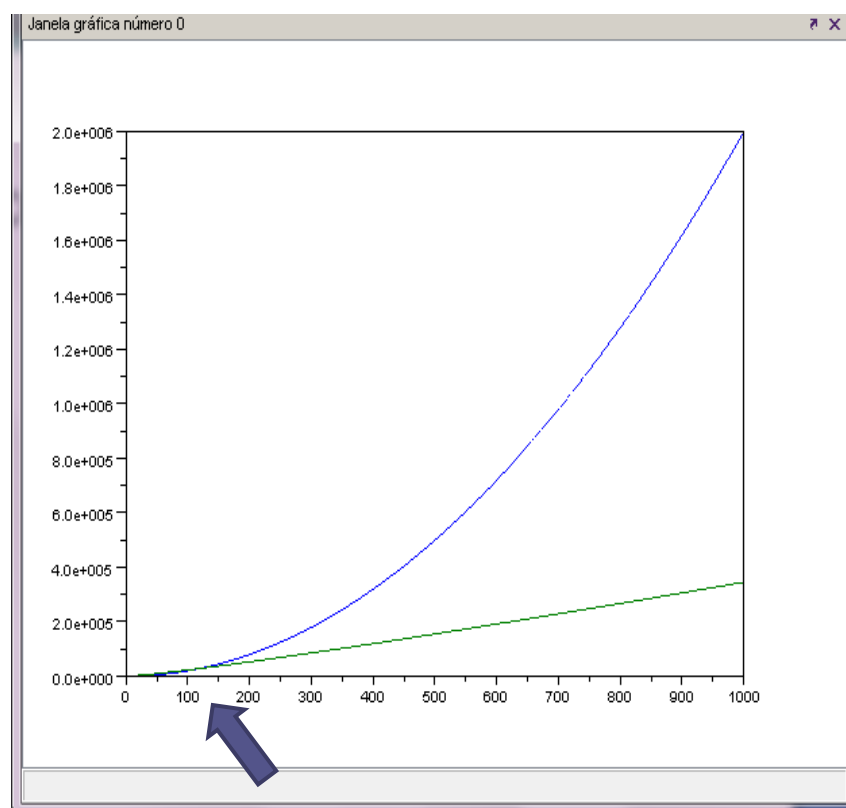


## Tendência

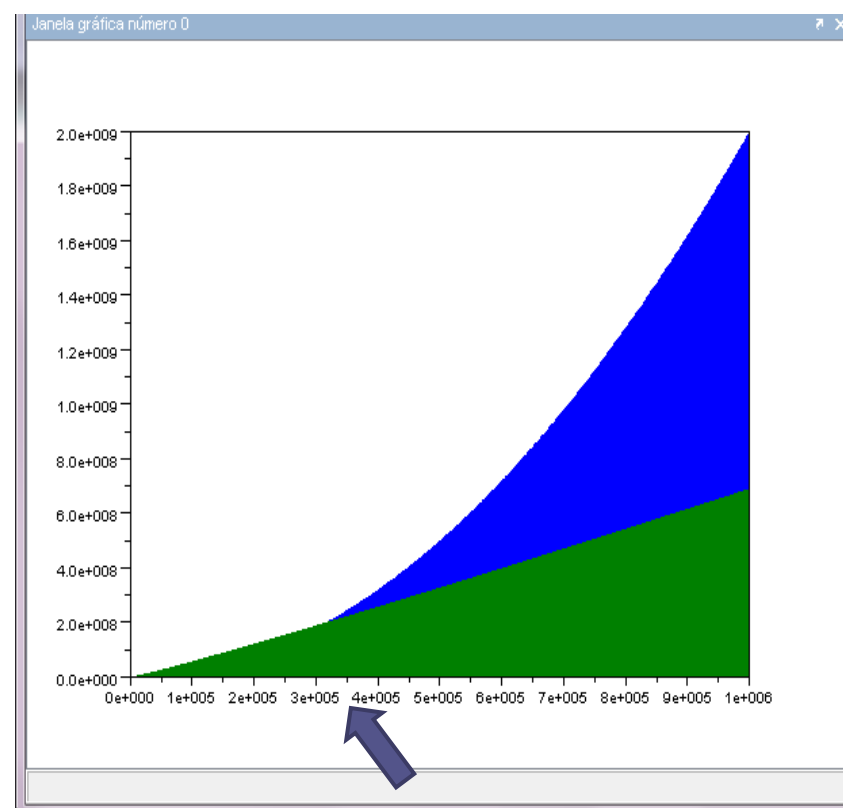


# Tecnologia só atrasa diferença...

## Iguais



## Mil vezes mais rápido



# Função de Complexidade

- Exercício. Forneça a função de complexidade de tempo do melhor caso, pior caso e caso médio para:

**function** sequencial(V,n, c)

    V[0] = c;

    i = n;

**while** V[i] != c:

        i = i - 1;

**return** i;

# Análise Assintótica - “A Idéia”

- Ignorar constantes dependentes de máquina
- Olhar a taxa de crescimento do tempo de execução, expressa em função do tamanho da instância de entrada.
  - Funções:  $\Theta$ ,  $O$ ,  $\Omega$ ,  $o$ ,  $\omega$



# Capturando essência da complexidade...

- Captura a tendência real do desempenho de um algoritmo, quando o tamanho da instância de entrada cresce.
- Em termos práticos:
  - Ignorar constantes e remover termos dominados da função de complexidade

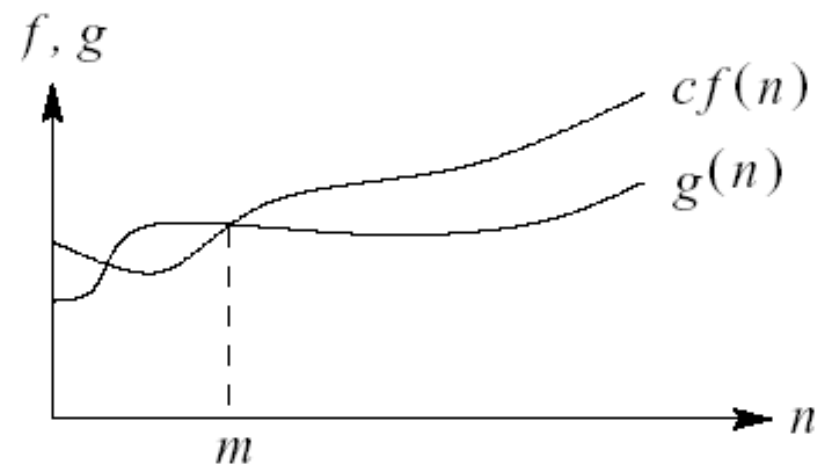
# Comportamento assintótico de funções

- O parâmetro  $n$  fornece uma medida da dificuldade para se resolver um problema, já que o tempo necessário para resolvê-lo cresce quando  $n$  cresce.
  - Para valores suficientemente pequenos de  $n$ , qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes.
  - A escolha do algoritmo não é um problema crítico para problemas de tamanho pequeno.
- A análise de algoritmos é realizada para valores grandes de  $n$ .
  - Estuda-se o comportamento assintótico das funções de custo.
  - O comportamento assintótico de  $f(n)$  representa o limite do comportamento do custo quando  $n$  cresce.

# Comportamento assintótico de funções

- A análise de um algoritmo geralmente conta com apenas algumas operações elementares.
  - A medida de custo ou medida de complexidade deve relatar o crescimento assintótico da(s) operação(ões) considerada(s).
- A seguinte definição relaciona o comportamento assintótico de duas funções distintas:

Uma função  $f(n)$  **domina assintoticamente** outra função  $g(n)$  se existem duas constantes positivas  $c$  e  $m$  tais que, para  $n \geq m$ , tem-se  $|g(n)| \leq c \times |f(n)|$ .



# Comportamento assintótico de funções

- Exemplo 1: Seja  $g(n) = n$  e  $f(n) = -n^2$ .
  - A função  $f(n)$  domina assintoticamente a função  $g(n)$ , já que  $|n| \leq 1 \times |-n^2|$  para  $n \geq 0$ .
  - A função  $g(n)$  não domina assintoticamente a função  $f(n)$ , já que  $|-n^2| > c \times |n|$  para todo  $n > 1$  e qualquer valor de  $c$  para algum valor de  $m$ .
- Exemplo 2: Seja  $g(n) = (n+1)^2$  e  $f(n) = n^2$ .
  - A função  $g(n)$  e  $f(n)$  dominam assintoticamente uma a outra, já que:
    - $|n^2| \leq 1 \times |(n+1)^2|$  para  $n \geq 0$ ;
    - $|(n+1)^2| \leq 4 \times |n^2|$  para  $n \geq 1$ .

# Comportamento assintótico de funções

- Para expressar que  $f(n)$  domina assintoticamente  $g(n)$ , uma notação sugerida por *Knuth* é  $g(n) = O(f(n))$ , onde se lê “ $g(n)$  é da ordem no máximo  $f(n)$ ”.
  - Assim, se o tempo de execução  $T(n)$  de um programa é  $O(n^2)$ , existem constantes  $c$  e  $m$  tais que, para valores de  $n \geq m$ ,  $T(n) \leq c \times n^2$ .
- **Definição** notação  $O$ : Uma função  $g(n)$  é  $O(f(n))$  se existem duas constantes positivas  $c$  e  $m$  tais que, para todo  $n \geq m$ , tem-se  $|g(n)| \leq c \times |f(n)|$ .
- Exemplo 1: Seja  $g(n) = (n+1)^2$ .
  - Logo  $g(n)$  é  $O(n^2)$ , já que  $|(n+1)^2| \leq 4 \times |n^2|$  para  $n \geq 1$ .

# Comportamento assintótico de funções

- Exemplo 2: Seja  $g(n) = 3n^3 + 2n^2 + n$ 
  - Logo  $g(n)$  é  $O(n^3)$ , já que  $|3n^3+2n^2+n| \leq 6 \times |n^3|$  para  $n \geq 0$ .
  - É importante notar que  $g(n)$  também é  $O(n^4)$ , porém tal fato é mais fraco do que dizer que  $g(n)$  é  $O(n^3)$ .
- Exemplo 3: Seja  $g(n) = n^2$ 
  - Claramente,  $g(n)$  é  $O(n^2)$ , já que  $|n^2| \leq 1 \times |n^2|$  para  $n \geq 0$ .
  - Porém,  $g(n)$  não é  $O(n)$ . Suponha que existam constantes  $c$  e  $m$  tais que para todo  $n \geq m$ ,  $n^2 \leq c \times n$ . Logo  $c \geq n$  para qualquer  $n \geq m$ . Mas, não existe  $c \geq n$  para todo  $n$ .
- Exemplo 4: Seja  $g(n) = \log_5 n$ 
  - $g(n)$  é  $O(\log n)$ , já que  $\log_b n$  difere de  $\log_c n$  pela const.  $\log_b c$ .
  - Como  $n = c^{\log_c n}$ , tomando  $\log_b$  em ambos os lados, tem-se:

$$\log_b n = \log_b c^{\log_c n} = \log_c n \times \log_b c.$$

- $\log_c n = \frac{\log_b n}{\log_b c}$

# Comportamento assintótico de funções

- Algumas operações realizadas com a notação  $O$  são:

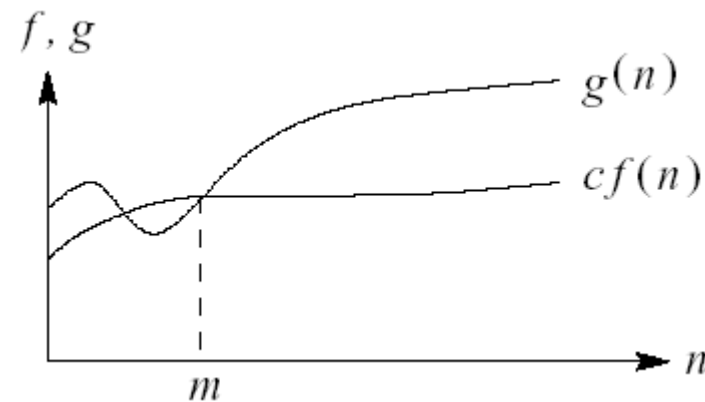
$$\begin{aligned}
 f(n) &= O(f(n)) \\
 c \times O(f(n)) &= O(f(n)) \quad c = \text{constante} \\
 O(f(n)) + O(f(n)) &= O(f(n)) \\
 O(O(f(n))) &= O(f(n)) \\
 O(f(n)) + O(g(n)) &= O(\max(f(n), g(n))) \\
 O(f(n))O(g(n)) &= O(f(n)g(n)) \\
 f(n)O(g(n)) &= O(f(n)g(n))
 \end{aligned}$$

- A regra da soma  $O(f(n)) + O(g(n))$  pode ser usada para calcular o tempo de execução de uma sequência de trechos de programa.
  - Suponha trechos cujo tempo de execução são  $O(n)$ ,  $O(n^2)$  e  $O(n \log n)$ . Assim, o tempo de execução do programa é  $O(n^2)$ .
- O produto de  $\lceil \log n + k + O(1/n) \rceil$  por  $\lceil n + O(\sqrt{n}) \rceil$  é:

$$n \log n + kn + O(\sqrt{n} \log n).$$

# Comportamento assintótico de funções

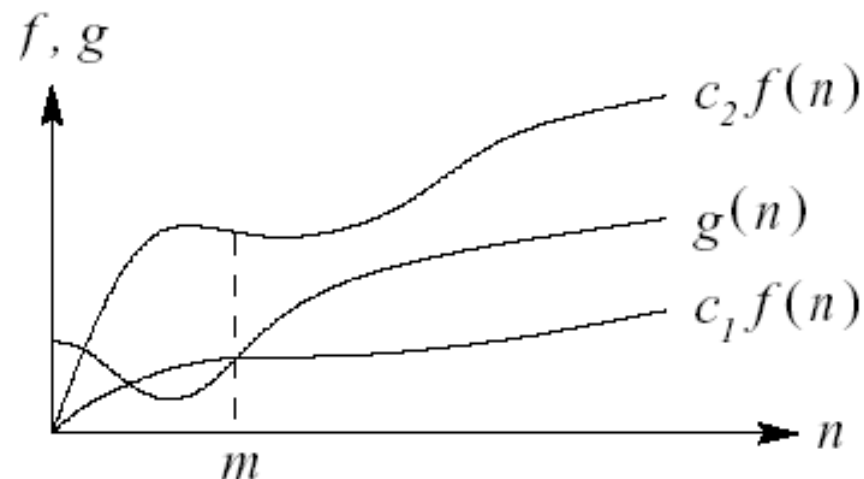
- Dizer que  $g(n)$  é  $O(f(n))$  significa que  $f(n)$  é um limite superior para  $g(n)$ . A notação  $\Omega$  especifica o limite inferior.
- **Definição** notação  $\Omega$ : Uma função  $g(n)$  é  $\Omega(f(n))$  se existem duas constantes positivas  $c$  e  $m$  tais que, para todo  $n \geq m$ , tem-se  $|g(n)| \geq c \times |f(n)|$ .
- Exemplo: Seja  $g(n) = 3n^3 + 2n^2$ 
  - Logo  $g(n)$  é  $\Omega(n^3)$ , já que  $|3n^3 + 2n^2| \geq 1 \times |n^3|$  para  $n \geq 0$ .





# Comportamento assintótico de funções

- **Definição** notação  $\Theta$ : Uma função  $g(n)$  é  $\Theta(f(n))$  se existirem constantes positivas  $c_1$ ,  $c_2$  e  $m$  tais que, para todo  $n \geq m$ , tem-se  $0 \leq c_1 \times f(n) \leq g(n) \leq c_2 \times f(n)$ .



- Para todo  $n \geq m$ ,  $g(n)$  é igual a  $f(n)$  a menos de uma constante. Nesse caso,  $f(n)$  é considerado um **limite assintótico firme**.

# Comportamento assintótico de funções

- Exemplo: Seja  $g(n) = 3n^2 + 2n + 2$ .
  - Para mostrar que  $g(n) = \Theta(n^2)$ , deve-se determinar valores para as constantes  $c_1$ ,  $c_2$  e  $m$  tais que:  
$$c_1 n^2 \leq 3n^2 + 2n + 2 \leq c_2 n^2, \text{ para } n \geq m$$
  - Para  $c_2$  podemos escolher 7, para  $n \geq 1$
  - Para  $c_1$  podemos escolher 1, para  $n \geq 0$
  - Logo, escolhendo  $c_1 = 1$ ,  $c_2 = 7$  e  $m = 1$ , é possível verificar que  $g(n) = \Theta(n^2)$ .
  - Outras constantes podem existir, mas o importante é que existe alguma escolha para as três constantes.

# Comportamento assintótico de funções

- A notação  $o$  é usada para definir um limite superior que não é assintoticamente firme.
- **Definição** notação  $o$ : Uma função  $g(n)$  é  $o(f(n))$  se, para qualquer constante  $c > 0$  existe  $m > 0$ , então  $0 \leq g(n) < c \times f(n)$  para todo  $n \geq m$ .
- Exemplo:  $2n = o(n^2)$ , mas  $2n^2 \neq o(n^2)$ .
- As notações  $O$  e  $o$  são similares.
  - A diferença é que em  $g(n) = O(f(n))$ , a expressão “ $0 \leq g(n) < c \times f(n)$ ” é válida para alguma constante  $c > 0$ , mas em  $g(n) = o(f(n))$ , tal expressão é válida para todas as constantes  $c > 0$ .
- Na notação  $o$ , a função  $g(n)$  tem um crescimento muito menor que  $f(n)$  quando  $n$  tende para infinito.
  - Alguns autores usam o seguinte limite para definição da notação  $o$ :

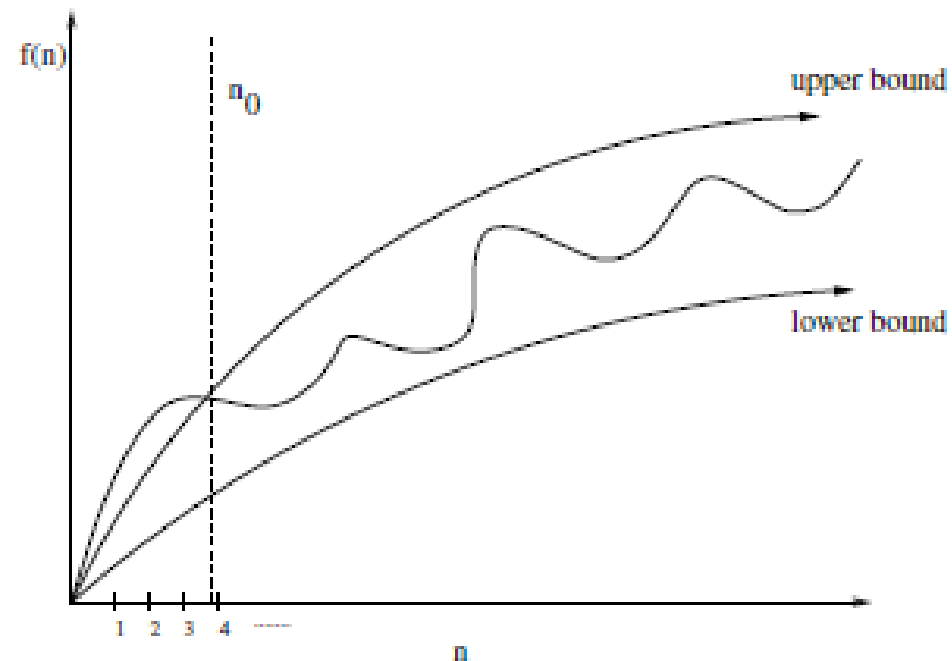
$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$$

# Comportamento assintótico de funções

- A notação  $\omega$  é usada para definir um limite inferior que não é assintoticamente firme.
- **Definição** notação  $\omega$ : Uma função  $g(n)$  é  $\omega(f(n))$  se, para qualquer constante  $c > 0$  existir  $m > 0$ , então  $0 \leq c \times f(n) < g(n)$  para todo  $n \geq m$ .
- Exemplo:  $n^2/2 = \omega(n)$ , mas  $n^2/2 \neq \omega(n^2)$ .
- As notações  $\Omega$  e  $\omega$  são similares.
  - A diferença é que em  $g(n) = \Omega(f(n))$ , a expressão “ $0 \leq c \times f(n) < g(n)$ ” é válida para alguma constante  $c > 0$ , mas em  $g(n) = \omega(f(n))$ , tal expressão é válida para todas as constantes  $c > 0$ .
- Na notação  $\omega$ , a função  $f(n)$  tem um crescimento muito menor que  $g(n)$  quando  $n$  tende para infinito.
  - Alguns autores usam o seguinte limite para definição da notação  $\omega$ :

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \infty$$

# Limites são mais suaves...



# Exercícios

- Em cada uma das situações mostre se  $f = O(g)$ ,  $f = \Omega(g)$ ,  $f = \Theta(g)$ .

□

f	g
$n-100$	$n-200$
$\log n$	$(\log n)^2$
$\log n$	$\log n^2$
$2^n$	$2^{n+1}$
$n!$	$2^n$
$2n^2 + 5n$	$n^2$
$2n^2+5n$	$n^3$

# Exercício

- Mostre que  $(n+2)^2 = \Theta(n^2)$
- Mostre que  $2^{2n} \neq O(2^n)$

# Análise de Comandos

- Atribuição:  $O(1)$
- Sequência: regra da soma

$$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$
$$\Omega(f(n)) + \Omega(g(n)) = \Omega(\max(f(n), g(n)))$$



# Análise de Comandos

- Decisão
  - Condição
  - Melhor (mínimo), pior (máximo), médio (?)

# Análise de Comandos

- Repetição: regra da multiplicação
  - Somatórios
  - Dependência de limite
  - While (pior, melhor, médio)

$$O(cf(n)) = O(f(n))$$

$$\Theta(f(n))\Theta(g(n)) = \Theta(f(n)g(n))$$

# Análise de Comandos

- Funções não-recursivas
  - Calcula complexidade da função e atribui ao comando de chamada
- Funções recursivas:
  - Relações de recorrência
  - Teorema mestre

# Multiplicação de Matrizes

```
for (i=1; i<=x; i++)  
    for (j=1; j<=y; j++) {  
        C[i][j] = 0;  
        for (k=1; k<=z; k++)  
            C[i][j] += A[i][k] * B[k][j];  
    }
```

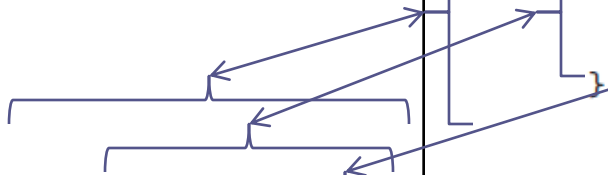
- Multiplicação de duas matrizes  $C_{x \times z} = A_{x \times y} \times B_{y \times z}$
- Matrizes começam no índice 1

# Multiplicação de Matrizes

```

for (i=1; i<=x; i++)
  for (j=1; j<=y; j++) {
    C[i][j] = 0;
    for (k=1; k<=z; k++)
      C[i][j] += A[i][k] * B[k][j];
  }

```



$$T(x, y, z) = \sum_{i=1}^x \sum_{j=1}^y \left( \Theta(1) + \sum_{k=1}^z \Theta(1) \right)$$

$$T(x, y, z) = \sum_{i=1}^x \sum_{j=1}^y (\Theta(1) + z\Theta(1))$$

$$T(x, y, z) = \sum_{i=1}^x \sum_{j=1}^y (\Theta(1) + \Theta(z)) = \sum_{i=1}^x \sum_{j=1}^y (\Theta(z))$$

$$T(x, y, z) = \sum_{i=1}^x y\Theta(z) = \sum_{i=1}^x \Theta(yz)$$

$$T(x, y, z) = x\Theta(yz) = \Theta(xyz)$$

# Pesquisa em texto

```
int findmatch(char *p, char *t)
{
    int i,j;                /* counters */
    int m, n;               /* string lengths */

    m = strlen(p);
    n = strlen(t);

    for (i=0; i<=(n-m); i=i+1) {
        j=0;
        while ((j<m) && (t[i+j]==p[j]))
            j = j+1;
        if (j == m) return(i);
    }

    return(-1);
}
```

para na

os  $m-1$   
ouco se isso

do while, se  
 $j = m-1$

# Pesquisa em texto

```

int findmatch(char *p, char *t)
{
    int i,j;                /* counters */
    int m, n;              /* string lengths */

    m = strlen(p);
    n = strlen(t);

    for (i=0; i<=(n-m); i=i+1) {
        j=0;
        while ((j<m) && (t[i+j]==p[j]))
            j = j+1;
        if (j == m) return(i);
    }

    return(-1);
}

```

$$T(n, m) = \Theta(n) + \Theta(m) + \sum_{i=0}^{n-m} \left( \Theta(1) + \sum_{j=1}^{m-1} \Theta(1) + \Theta(1) \right) + \Theta(1)$$

$$T(n, m) = \Theta(n) + \sum_{i=0}^{n-m} (\Theta(1) + (m-1)\Theta(1))$$

$$T(n, m) = \Theta(n) + \sum_{i=0}^{n-m} (\Theta(1) + \Theta(m)) = \Theta(n) + \sum_{i=0}^{n-m} (\Theta(m))$$

$$T(n, m) = \Theta(n) + (n - m + 1)\Theta(m) = \Theta(n) + \Theta(nm - m^2 + m)$$

$$T(n, m) = \Theta(n) + \Theta(nm) = \Theta(nm)$$

# Análise com chamadas de função

```
function fib2(n)  
if n = 0 return 0  
create an array f [0...n]  
f [0] = 0, f [1] = 1  
for i = 2...n:  
    f [i] = f [i - 1] + f [i - 2]  
return f [n]
```

$$T_1(n) = \Theta(n)$$

function escreveFibonacci( *n*)

```
1   for i = 1..n:  
2       val = fib2(i);
```

$$\begin{aligned} T_2(n) &= \sum_{i=1}^n T_1(i) \\ &= \sum_{i=1}^n \Theta(i) \\ &= \Theta\left(\frac{n(n+1)}{2}\right) \\ &= \Theta(n^2) \end{aligned}$$



# Algoritmos recursivos

function pesquisaBinaria( *chave*, *esq*, *dir*, *a*[1..*n*] )

```
1  if esq > dir:  
2      return no;  
3  meio = (dir+esq)/2;  
4  if (a[meio] == chave):  
5      return yes;  
6  if (a[meio] > chave):  
7      return pesquisaBinaria(chave, esq, meio-1, a)  
8  else:  
9      return pesquisaBinaria(chave, meio+1, dir, a)
```


# Complexidade - Recorrência

function pesquisaBinaria( *chave*, *esq*, *dir*, *a*[1..*n*] )

```
1  if esq > dir:  
2      return no;  
3  meio = (dir+esq)/2;  
4  if (a[meio] == chave):  
5      return yes;  
6  if (a[meio] > chave):  
7      return pesquisaBinaria(chave, esq, meio-1, a)  
8  else:  
9      return pesquisaBinaria(chave, meio+1, dir, a)
```

$$T(n) = \begin{cases} 1 & n = 1 \\ 4 + T\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

# Recorrência: terminologia

$$T(n) = \begin{cases} 1 & n = 1 \\ 4 + T\left(\frac{n}{2}\right) & n > 1 \end{cases}$$


Base

$$T(n) = 4\lceil \log n \rceil + 1$$



Forma Fechada da  
Função de Complexidade

# Iteração

$$T(n) = 4 + T\left(\frac{n}{2}\right)$$

$$T(n) = 4 + \left(4 + T\left(\frac{n}{4}\right)\right) = 2 \times 4 + T\left(\frac{n}{2^2}\right)$$

$$T(n) = 2 \times 4 + \left(4 + T\left(\frac{n}{2^3}\right)\right) = 3 \times 4 + T\left(\frac{n}{2^3}\right)$$

...

$$T(2^k) = k \times 4 + T\left(\frac{n}{2^k}\right) = \log n \times 4 + T(1)$$

$$T(n) = \Theta(\log n)$$

$$\begin{aligned} \frac{n}{2^k} &= 1 \\ n &= 2^k \\ k &= \lg n \end{aligned}$$

# Exercício

- Faça um algoritmo que informe o  $n$ -ésimo número de Fibonacci em  $O(\log n)$ .