

**Pontifícia Universidade Católica de Minas Gerais  
Instituto de Ciências Exatas e Informática  
Departamento de Ciência da Computação  
Curso de Ciência da Computação**

# **Projeto e Análise de Algoritmos**

## **Parte 1**

**Raquel Mini  
raquelmini@pucminas.br**

---

# O que é um algoritmo?

- Qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como entrada e produz algum valor ou conjunto de valores como saída
- Sequência de passos computacionais que transformam a entrada na saída
- Sequência de ações executáveis para a obtenção de uma solução para um determinado tipo de problema
- Descrição de um padrão de comportamento, expresso em termos de um conjunto finito de ações
- Sequência não ambígua de instruções que é executada até que determinada condição se verifique

# Algoritmo correto X incorreto

- Um algoritmo é **correto** se, para cada instância de entrada, ele para com a saída correta
- Um algoritmo **incorreto** pode não parar em algumas instâncias de entrada, ou então pode parar com outra resposta que não a desejada

# Algoritmo eficiente X ineficiente

- Algoritmos **eficientes** são os que executam em tempo polinomial
- Algoritmos que necessitam de tempo superpolinomial são chamados de **ineficientes**

# Problema tratável X intratável

- Problemas que podem ser resolvidos por algoritmo de tempo polinomial são chamados de **tratáveis**
- Problemas que exigem tempo superpolinomial são chamados de **intratáveis**

## Tratabilidade

# Problema decidível X indecidível

- Um problema é **decidível** se existe algoritmo para resolvê-lo
- Um problema é **indecidível** se não existe algoritmo para resolvê-lo

## Decidibilidade

# Análise de algoritmos

- Analisar a complexidade computacional de um algoritmo significa prever os recursos de que o mesmo necessitará:
  - Memória
  - Largura de banda de comunicação
  - Hardware
  - **Tempo de execução**
- Geralmente existe mais de um algoritmo para resolver um problema
- A análise de complexidade computacional é fundamental no processo de definição de algoritmos mais eficientes para a sua solução
- Em geral, o tempo de execução cresce com o tamanho da entrada

# Porque estudar análise de algoritmos?

- O tempo de computação e o espaço na memória são recursos limitados
  - Os computadores podem ser rápidos, mas não são infinitamente rápidos
  - A memória pode ser de baixo custo, mas é finita e não é gratuita
- Os recursos devem ser usados de forma sensata, e algoritmos eficientes em termos de tempo e espaço devem ser projetados
- Com o aumento da velocidade dos computadores, torna-se cada vez mais importante desenvolver algoritmos mais eficientes, devido ao aumento constante do tamanho dos problemas a serem resolvidos



# Porque estudar análise de algoritmos?

- Suponha que para resolver um determinado problema você tem disponível um **algoritmo exponencial** ( $2^n$ ) e um computador capaz de executar  $10^4$  operações por segundo

		2 <sup>n</sup> na máquina 10 <sup>4</sup>
	tempo (s)	tamanho
	0,10	10
	1	13
1 minuto	60	19
1 hora	3.600	25
1 dia	86.400	30
1 ano	31.536.000	38

# Porque estudar análise de algoritmos?

- Compra de um novo computador capaz de executar  $10^9$  operações por segundo

		2 <sup>n</sup> na máquina 10 <sup>4</sup>	2 <sup>n</sup> na máquina 10 <sup>9</sup>
tempo (s)		tamanho	tamanho
	0,10	10	27
	1	13	30
	60	19	36
1 minuto	60	19	36
1 hora	3.600	25	42
1 dia	86.400	30	46
1 ano	31.536.000	38	55

*Aumento na velocidade computacional tem pouco efeito no tamanho das instâncias resolvidas por algoritmos ineficientes*

# Porque estudar análise de algoritmos?

- **Investir em algoritmo:**
  - Você encontrou um algoritmo quadrático ( $n^2$ ) para resolver o problema

		2 <sup>n</sup> na máquina 10 <sup>4</sup>	2 <sup>n</sup> na máquina 10 <sup>9</sup>	n <sup>2</sup> na máquina 10 <sup>4</sup>	n <sup>2</sup> na máquina 10 <sup>9</sup>
tempo (s)		tamanho	tamanho	tamanho	tamanho
1 minuto	0,10	10	27	32	10.000
	1	13	30	100	31.623
	60	19	36	775	244.949
1 hora	3.600	25	42	6.000	1.897.367
1 dia	86.400	30	46	29.394	9.295.160
1 ano	31.536.000	38	55	561.569	177.583.783

*Novo algoritmo oferece uma melhoria maior  
que a compra da nova máquina*

# Porque estudar projeto de algoritmos?

- Algum dia você poderá encontrar um problema para o qual não seja possível descobrir prontamente um algoritmo publicado
- É necessário estudar técnicas de projeto de algoritmos, de forma que você possa desenvolver algoritmos por conta própria, mostrar que eles fornecem a resposta correta e entender sua eficiência

# Exercício

1. Para cada função  $f(n)$  e cada tempo  $t$  na tabela a seguir, determine o maior tamanho  $n$  de um problema que pode ser resolvido no tempo  $t$ , supondo-se que o algoritmo para resolver o problema demore  $f(n)$  segundos

	1 seg.	1 min.	1 hora	1 dia	1 mês	1 ano	1 século
$\log n$							
$\sqrt{n}$							
$n$							
$n \log n$							
$n^2$							
$n^3$							
$2^n$							
$n!$							

# Medida do Tempo de Execução de um Programa

Ziviani – págs. 1 até 11

Cormen – págs. 3 até 20

# Tipos de problemas na análise de algoritmos

- Análise de um algoritmo particular
- Análise de uma classe de algoritmos

# Análise de um algoritmo particular

- Qual é o custo de usar um dado algoritmo para resolver um problema específico?
- Características que devem ser investigadas:
  - Análise do número de vezes que cada parte do algoritmo deve ser executada
  - Estudo da quantidade de memória necessária



# Análise de uma classe de algoritmos

- Qual é o algoritmo de menor custo possível para resolver um problema particular?
- Toda uma família de algoritmos é investigada
- Procura-se identificar um que seja o melhor possível
- Colocam-se **limites** para a complexidade computacional dos algoritmos pertencentes à classe

# Custo de um algoritmo

- Determinando o menor custo possível para resolver problemas de uma dada classe, temos a medida da dificuldade inerente para resolver o problema
- Quando o custo de um algoritmo é igual ao menor custo possível, o algoritmo é **ótimo** para a medida de custo considerada
- Podem existir vários algoritmos para resolver o mesmo problema
- Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado

# Função de complexidade

- Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou **função de complexidade**  $T$
- $T(n)$  é a medida do tempo necessário para executar um algoritmo para um problema de tamanho  $n$
- Função de **complexidade de tempo**:  $T(n)$  mede o tempo necessário para executar um algoritmo para um problema de tamanho  $n$
- Função de **complexidade de espaço**:  $T(n)$  mede a memória necessária para executar um algoritmo para um problema de tamanho  $n$
- Utilizaremos  $T$  para denotar uma função de complexidade de tempo daqui para a frente
- Na realidade, a complexidade de tempo não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada

# Exemplo: Maior elemento

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros  $A[1..n]$ ,  $n \geq 1$

```
function Max (var A: Vetor): integer;  
var i, Temp: integer;  
begin  
    Temp := A[1];  
    for i:=2 to n do if Temp < A[i] then Temp := A[i];  
    Max := Temp;  
end;
```

- Seja  $T$  uma função de complexidade tal que  $T(n)$  seja o número de comparações entre os elementos de  $A$ , se  $A$  contiver  $n$  elementos
- Logo  $T(n) = n - 1$  para  $n \geq 1$
- Vamos provar que o algoritmo apresentado no programa acima é **ótimo**

# Exemplo: Maior elemento

- **Teorema:** Qualquer algoritmo para encontrar o maior elemento de um conjunto com  $n$  elementos,  $n \geq 1$ , faz pelo menos  $n - 1$  comparações
- **Prova:** Cada um dos  $n - 1$  elementos tem de ser mostrado, por meio de comparações, que é menor que algum outro elemento
- Logo  $n - 1$  comparações são necessárias
- O teorema acima nos diz que, se o número de comparações for utilizado para medida de custo, então a função  $\text{Max}$  do programa anterior é ótima

# Tamanho da entrada de dados

- A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada de dados
- É comum considerar o tempo de execução de um programa como uma função do tamanho da entrada
- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada
- No caso da função  $\text{Max}$  do programa do exemplo, o custo é uniforme sobre todos os problemas de tamanho  $n$
- Já para um algoritmo de ordenação isso não ocorre: se os dados de entrada já estiverem quase ordenados, então o algoritmo pode ter que trabalhar menos

# Melhor caso, pior caso e caso médio

- **Melhor caso:**
  - Menor tempo de execução sobre todas as entradas de tamanho  $n$
- **Pior caso:**
  - Maior tempo de execução sobre todas as entradas de tamanho  $n$
  - Se  $T$  é uma função de complexidade baseada na análise de pior caso, o custo de aplicar o algoritmo nunca é maior do que  $T(n)$
- **Caso médio (ou caso esperado):**
  - Média dos tempos de execução de todas as entradas de tamanho  $n$

# Melhor caso, pior caso e caso médio

- Na análise do caso esperado, supõe-se uma **distribuição de probabilidades** sobre o conjunto de entradas de tamanho  $n$  e o custo médio é obtido com base nessa distribuição
- A análise do caso médio é geralmente muito mais difícil de obter do que as análises do melhor e do pior caso
- É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são igualmente prováveis
- Na prática isso nem sempre é verdade



# Exemplo: Registros de um arquivo

- Considere o problema de acessar os registros de um arquivo
- Cada registro contém uma chave única que é utilizada para recuperar registros do arquivo
- O problema: dada uma chave qualquer, localize o registro que contenha esta chave
- O algoritmo de pesquisa mais simples é o que faz a pesquisa sequencial

# Exemplo: Registros de um arquivo

- Seja  $T$  uma função de complexidade tal que  $T(n)$  é o número de registros consultados no arquivo (número de vezes que a chave de consulta é comparada com a chave de cada registro)
  - Melhor caso:  $T(n) = 1$  (registro procurado é o primeiro consultado)
  - Pior caso:  $T(n) = n$  (registro procurado é o último consultado ou não está presente no arquivo)
  - Caso médio:  $T(n) = \frac{(n+1)}{2}$

# Exemplo: Registros de um arquivo

- No estudo do caso médio, vamos considerar que toda pesquisa recupera um registro
- Se  $p_i$  for a probabilidade de que o  $i$ -ésimo registro seja procurado, e considerando que para recuperar o  $i$ -ésimo registro são necessárias  $i$  comparações, então

$$T(n) = (1 \times p_1) + (2 \times p_2) + (3 \times p_3) + \cdots + (n \times p_n)$$

- Para calcular  $T(n)$  basta conhecer a distribuição de probabilidades  $p_i$
- Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então  $p_i = 1/n, 1 \leq i \leq n$

- Neste caso 
$$T(n) = \frac{1}{n} (1 + 2 + 3 + \cdots + n) = \frac{1}{n} \left( \frac{n(n+1)}{2} \right) = \frac{n+1}{2}$$

- A análise do caso esperado revela que uma pesquisa com sucesso examina aproximadamente metade dos registros

# Exercício

2. No problema de acessar os registros de um arquivo, seja  $T$  uma função de complexidade tal que  $T(n)$  é o número de registros consultados no arquivo. Seja  $q$  a probabilidade de que uma pesquisa seja realizada com sucesso (chave procurada se encontra no arquivo) e  $(1 - q)$  a probabilidade da pesquisa sem sucesso (chave procurada não se encontra no arquivo). Considere também que nas pesquisas com sucesso todos os registros são igualmente prováveis. Encontre a função de complexidade para o caso médio.

# Exemplo: Maior e menor elementos (1)

- Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros  $A[1..n]$ ,  $n \geq 1$
- Um algoritmo simples pode ser derivado do algoritmo apresentado no programa para achar o maior elemento

```
procedure MaxMin1 (var A: Vetor; var Max, Min: integer);  
var i: integer;  
begin  
  Max := A[1]; Min := A[1];  
  for i := 2 to n do  
    begin  
      if A[i] > Max then Max := A[i]; {Testa se A[i] contém o maior elemento}  
      if A[i] < Min then Min := A[i]; {Testa se A[i] contém o menor elemento}  
    end;  
  end;  
end;
```

- Seja  $T(n)$  o número de comparações entre os elementos de  $A$ , se  $A$  tiver  $n$  elementos
- Logo  $T(n) = 2(n - 1)$ , para  $n > 0$ , para o melhor caso, pior caso e caso médio.

# Exemplo: Maior e menor elementos (2)

- MaxMin1 pode ser facilmente melhorado:
  - a comparação  $A[i] < \text{Min}$  só é necessária quando o resultado da comparação  $A[i] > \text{Max}$  for falso

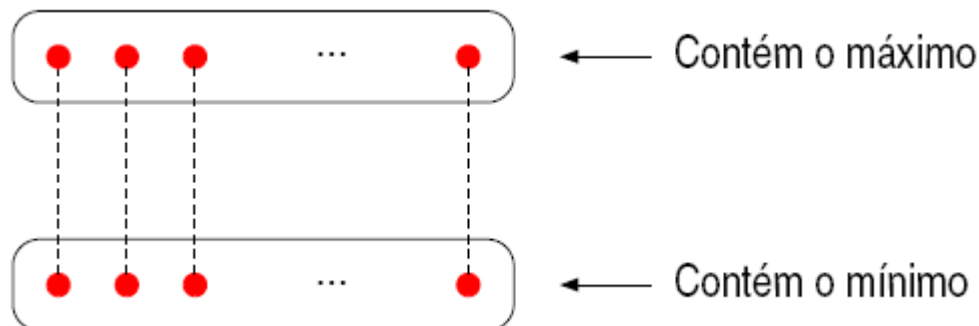
```
procedure MaxMin2 (var A: Vetor; var Max, Min: integer);
var i: integer;
begin
  Max := A[1];  Min := A[1];
  for i := 2 to n do
    if A[i] > Max
    then Max := A[i]
    else if A[i] < Min then Min := A[i];
  end;
```

# Exemplo: Maior e menor elementos (2)

- Para a nova implementação temos:
  - Melhor caso:  $T(n) = n - 1$  (quando os elementos estão em ordem crescente)
  - Pior caso:  $T(n) = 2(n - 1)$  (quando o maior elemento está na 1ª posição)
  - Caso médio:  $T(n) = \frac{3n}{2} - \frac{3}{2}$
- Caso médio:
  - $A[i]$  é maior do que  $\text{Max}$  a metade das vezes
  - Logo,  $T(n) = n - 1 + \frac{n - 1}{2} = \frac{3n}{2} - \frac{3}{2}$ , para  $n > 0$

# Exemplo: Maior e menor elementos (3)

- Considerando o número de comparações realizadas, existe a possibilidade de obter um algoritmo mais eficiente:
  1. Compare os elementos de  $A$  aos pares, separando-os em dois subconjuntos (maiores em um e menores em outro), a um custo de  $\lceil n/2 \rceil$  comparações
  2. O máximo é obtido do subconjunto que contém os maiores elementos, a um custo de  $\lceil n/2 \rceil - 1$  comparações
  3. O mínimo é obtido do subconjunto que contém os menores elementos, a um custo de  $\lceil n/2 \rceil - 1$  comparações





# Exemplo: Maior e menor elementos (3)

```
procedure MaxMin3(var A: Vetor;  
                  var Max, Min: integer);  
var i,  
    FimDoAnel: integer;  
begin  
    {Garante uma qte par de elementos no vetor para evitar caso de exceção}  
    if (n mod 2) > 0  
    then begin  
        A[n+1] := A[n];  
        FimDoAnel := n;  
    end  
    else FimDoAnel := n-1;  
  
    {Determina maior e menor elementos iniciais}  
    if A[1] > A[2]  
    then begin  
        Max := A[1]; Min := A[2];  
    end  
    else begin  
        Max := A[2]; Min := A[1];  
    end;  
end;
```

# Exemplo: Maior e menor elementos (3)

```
i:= 3;
while i <= FimDoAnel do
  begin
    {Compara os elementos aos pares}
    if A[i] > A[i+1]
    then begin
      if A[i] > Max then Max := A[i];
      if A[i+1] < Min then Min := A[i+1];
    end
    else begin
      if A[i] < Min then Min := A[i];
      if A[i+1] > Max then Max := A[i+1];
    end;
    i:= i + 2;
  end;
end;
```

# Exemplo: Maior e menor elementos (3)

- Os elementos de  $A$  são comparados dois a dois e os elementos maiores são comparados com  $Max$  e os elementos menores são comparados com  $Min$
- Quando  $n$  é ímpar, o elemento que está na posição  $A[n]$  é duplicado na posição  $A[n+1]$  para evitar um tratamento de exceção
- Para esta implementação,  $T(n) = \frac{n}{2} + \frac{n-2}{2} + \frac{n-2}{2} = \frac{3n}{2} - 2$   
para  $n > 0$ , para o melhor caso, pior caso e caso médio

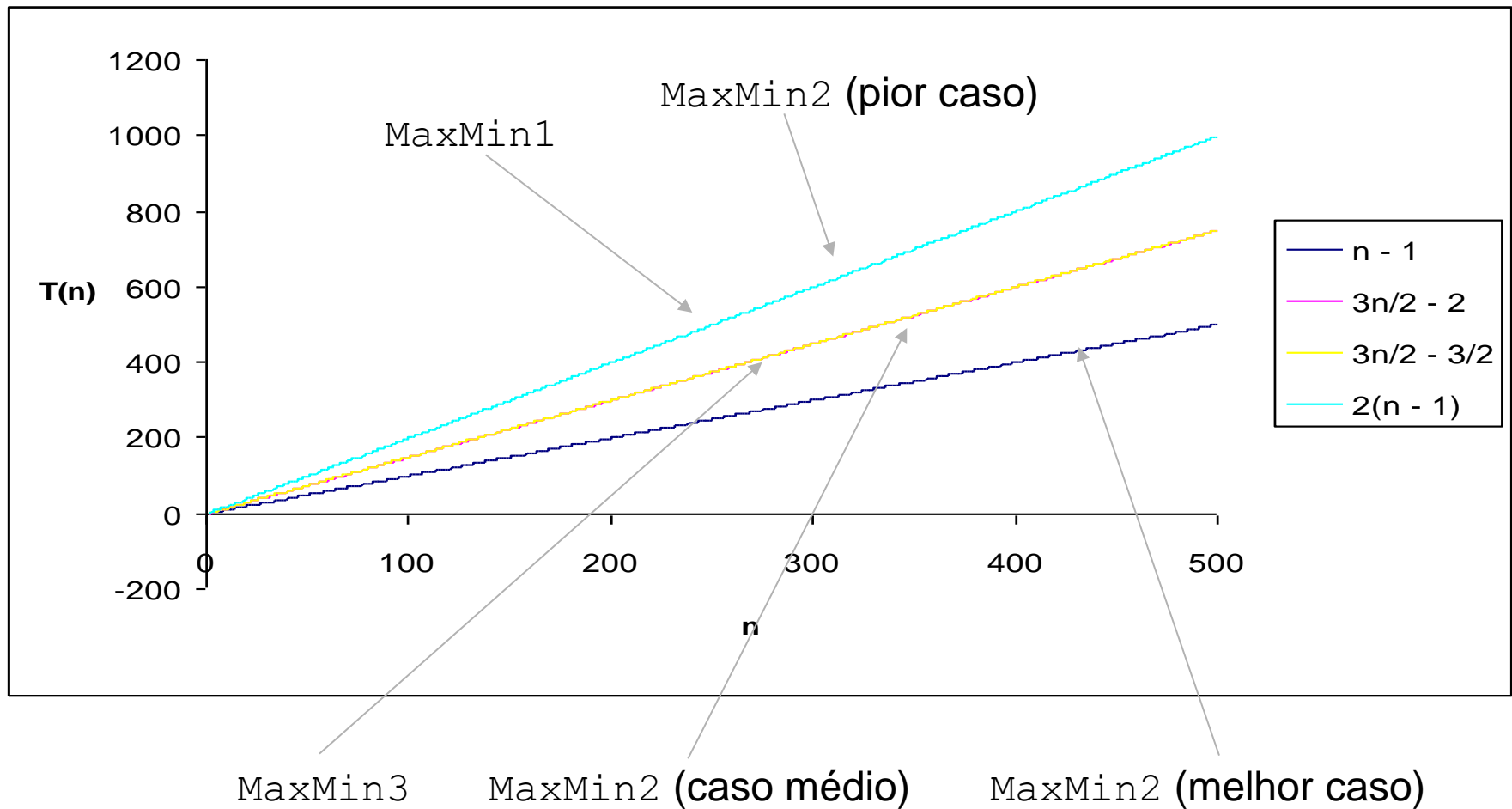
# Comparação entre os algoritmos

## MaxMin1, MaxMin2 e MaxMin3

- A tabela abaixo apresenta uma comparação entre os algoritmos dos programas MaxMin1, MaxMin2 e MaxMin3, considerando o número de comparações como medida de complexidade
- Os algoritmos MaxMin2 e MaxMin3 são superiores ao algoritmo MaxMin1 de forma geral.
- O algoritmo MaxMin3 é superior ao algoritmo MaxMin2 com relação ao pior caso e bastante próximo quanto ao caso médio

Os três algoritmos	T (n)		
	Melhor caso	Pior caso	Caso médio
MaxMin1	$2(n - 1)$	$2(n - 1)$	$2(n - 1)$
MaxMin2	$n - 1$	$2(n - 1)$	$3n/2 - 3/2$
MaxMin3	$3n/2 - 2$	$3n/2 - 2$	$3n/2 - 2$

# Comparação entre os algoritmos MaxMin1, MaxMin2 e MaxMin3



# Exercício

3. Apresente a função de complexidade de tempo para os algoritmos abaixo, indicando em cada caso qual é a operação relevante:

a)

```
ALG1 ()  
  for i ← 1 to 2 do  
    for j ← i to n do  
      for k ← i to j do  
        temp ← temp + i + j + k
```

b)

```
INSERTION-SORT(A)  
  for j ← 2 to n do  
    chave ← A[j]  
    i ← j - 1  
    A[0] ← chave //sentinela  
    while A[i] > chave do  
      A[i+1] ← A[i]  
      i ← i-1  
    A[i+1] ← chave
```

# Exercício

c)

```
BUBBLE-SORT (A)
for i  $\leftarrow$  1 to n do
    for j  $\leftarrow$  n downto i+1 do
        if A[j] < A[j-1] then
            A[j]  $\leftrightarrow$  A[j-1]
```

d)

```
SELECTION-SORT (A)
for i  $\leftarrow$  1 to n-1 do
    Min  $\leftarrow$  i
    for j  $\leftarrow$  i+1 to n do
        if A[j] < A[Min] then
            Min  $\leftarrow$  j
    A[Min]  $\leftrightarrow$  A[i]
```

# Comportamento Assintótico

Ziviani – págs. 11 até 19

Cormen – págs. 32 até 49

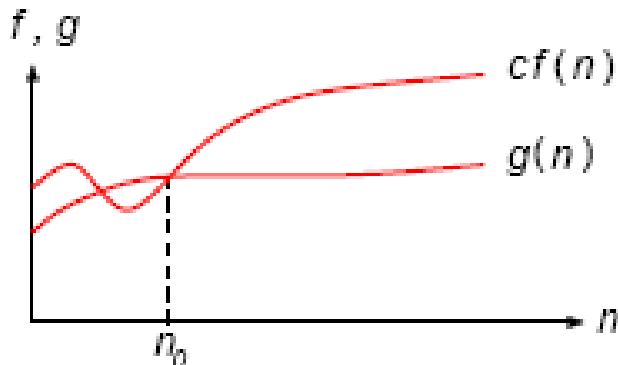


# Comportamento assintótico de funções

- O parâmetro  $n$  fornece uma medida da dificuldade para se resolver o problema
- 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
- Logo, a análise de algoritmos é realizada para valores grandes de  $n$
- Estuda-se o comportamento assintótico das **funções de custo** (comportamento de suas funções de custo para valores grandes de  $n$ )
- Para entradas grandes o bastante, as constantes multiplicativas e os termos de mais baixa ordem de um tempo de execução podem ser ignorados

# Dominação assintótica

- A análise de um algoritmo geralmente conta com apenas algumas operações elementares
- A medida de custo ou medida de complexidade relata o crescimento assintótico da operação considerada
- **Definição:** Uma função  $f(n)$  **domina assintoticamente** outra função  $g(n)$  se existem duas constantes positivas  $c$  e  $n_0$  tais que, para  $n \geq n_0$ , temos  $|g(n)| \leq c \times |f(n)|$



- Exemplo:
  - Sejam  $g(n) = (n+1)^2$  e  $f(n) = n^2$
  - As funções  $g(n)$  e  $f(n)$  dominam assintoticamente uma a outra, já que
  - $|(n+1)^2| \leq 4|(n^2)|$  para  $n \geq 1$  e
  - $|(n^2)| \leq |(n+1)^2|$  para  $n \geq 0$

# Como medir o custo de execução de um algoritmo?

- **Função de Custo ou Função de Complexidade**
  - $T(n)$  = medida de custo necessário para executar um algoritmo para um problema de tamanho  $n$
  - Se  $T(n)$  é uma medida da quantidade de tempo necessário para executar um algoritmo para um problema de tamanho  $n$ , então  $T$  é chamada função de complexidade de tempo de algoritmo
  - Se  $T(n)$  é uma medida da quantidade de memória necessária para executar um algoritmo para um problema de tamanho  $n$ , então  $T$  é chamada função de complexidade de espaço de algoritmo
- **Observação: tempo não é tempo!**
  - É importante ressaltar que a complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada

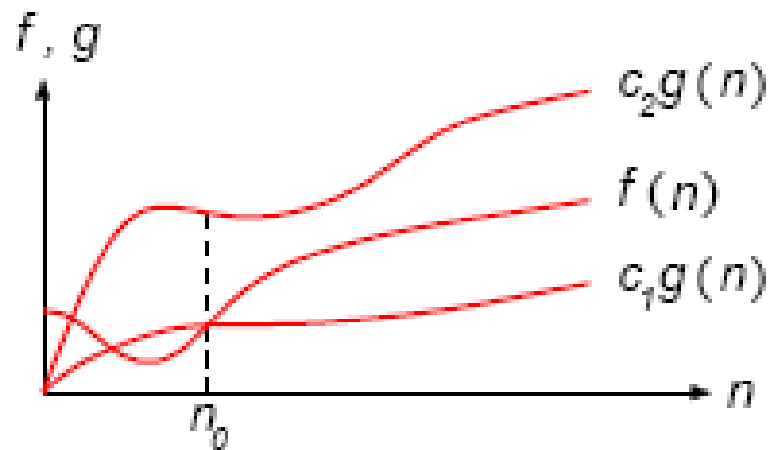
# Custo assintótico de funções

- É interessante comparar algoritmos para valores grandes de  $n$
- O custo assintótico de uma função  $T(n)$  representa o limite do comportamento de custo quando  $n$  cresce
- Em geral, o custo aumenta com o tamanho  $n$  do problema
- Observação:
  - Para valores pequenos de  $n$ , mesmo um algoritmo ineficiente não custa muito para ser executado

# Notação assintótica de funções

- Existem três notações principais na análise de assintótica de funções:
  - Notação  $\Theta$
  - Notação  $O$  (“ $O$ ” grande)
  - Notação  $\Omega$

# Notação $\Theta$



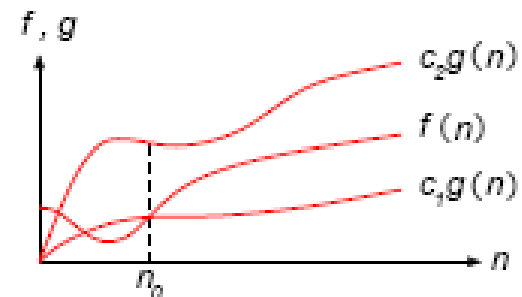
$$f(n) = \Theta(g(n))$$

# Notação $\Theta$

- A notação  $\Theta$  limita a função por fatores constantes
- Escreve-se  $f(n) = \Theta(g(n))$ , se existirem constantes positivas  $c_1$ ,  $c_2$  e  $n_0$  tais que para  $n \geq n_0$ , o valor de  $f(n)$  está sempre entre  $c_1g(n)$  e  $c_2g(n)$  inclusive
- Neste caso, pode-se dizer que  $g(n)$  é um limite assintótico firme (em inglês, *asymptotically tight bound*) para  $f(n)$

$$f(n) = \Theta(g(n)), \exists c_1 > 0, c_2 > 0 \text{ e } n_0 \mid$$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \quad \forall n \geq n_0$$



# Notação $\Theta$ : Exemplo

- Mostre que  $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

Para provar esta afirmação, devemos achar constantes  $c_1 > 0$ ,  $c_2 > 0$ ,  $n_0 > 0$ , tais que:

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

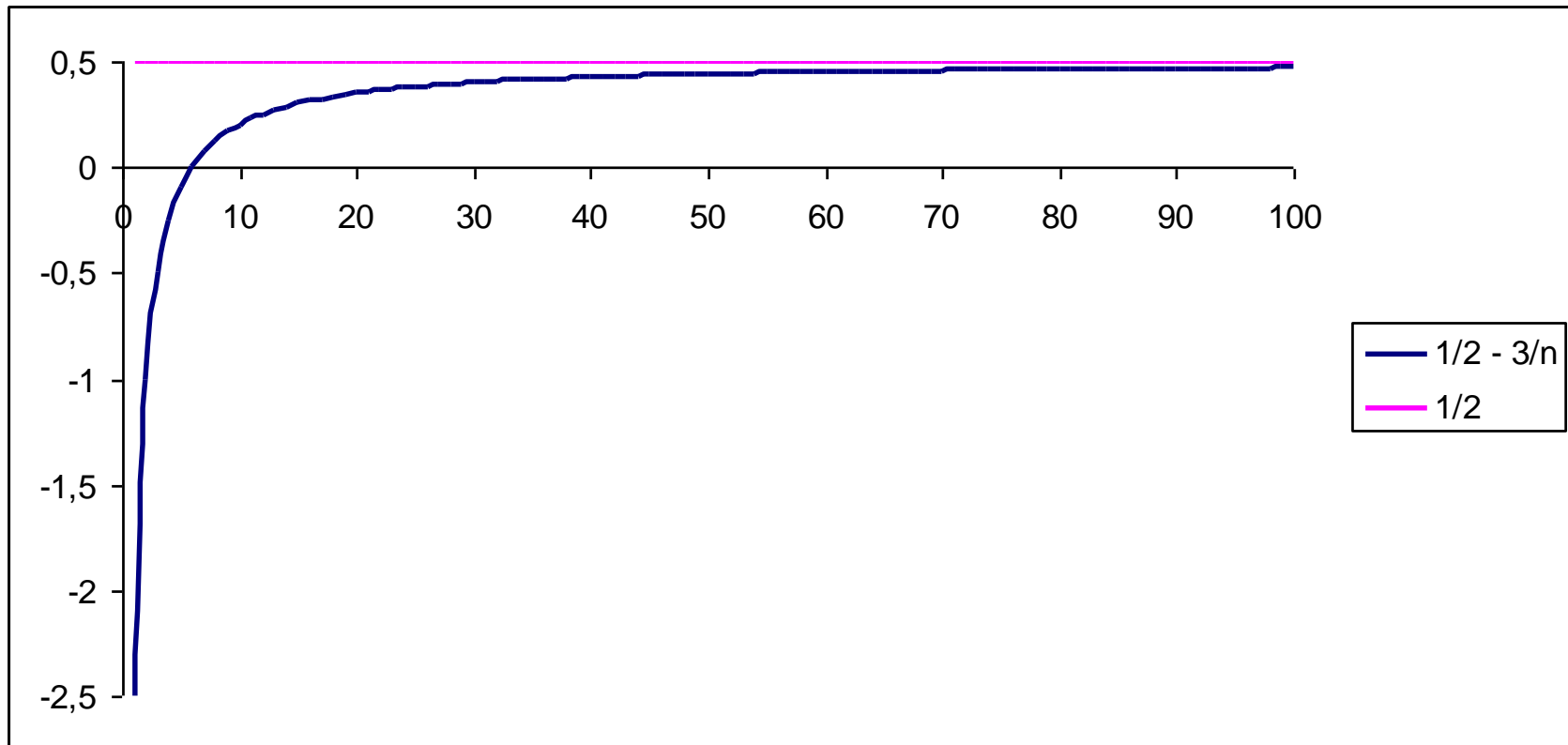
para todo  $n \geq n_0$

Se dividirmos a expressão acima por  $n^2$  temos:

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$



# Notação $\Theta$ : Exemplo



# Notação $\Theta$ : Exemplo

- A inequação mais a direita será sempre válida para qualquer valor de  $n \geq 1$  ao escolhermos  $c_2 \geq \frac{1}{2}$
- Da mesma forma, a inequação mais a esquerda será sempre válida para qualquer valor de  $n \geq 7$  ao escolhermos  $c_1 \leq \frac{1}{14}$
- Assim, ao escolhermos  $c_1 = 1/14$ ,  $c_2 = 1/2$  e  $n_0 = 7$ , podemos verificar que  $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
- Note que existem outras escolhas para as constantes  $c_1$  e  $c_2$ , mas o fato importante é que a escolha existe
- Note também que a escolha destas constantes depende da função  $\frac{1}{2}n^2 - 3n$
- Uma função diferente pertencente a  $\Theta(n^2)$  irá provavelmente requerer outras constantes

# Exercício

4. Prove que:

a)  $2n^2 + n = \Theta(n^2)$

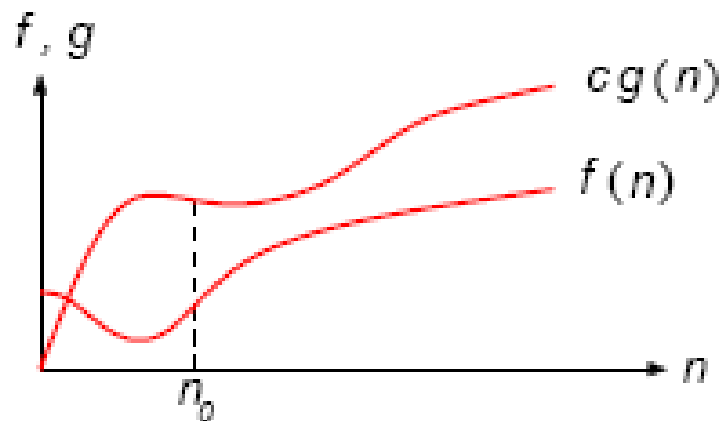
b)  $3n^3 + 2n^2 + n = \Theta(n^3)$

c)  $\log_5^n = \Theta(\log n)$

d)  $7n \log n + n = \Theta(n \log n)$

5. Usando a definição formal de  $\Theta$ , prove que  $6n^3 \neq \Theta(n^2)$ .

# Notação O



$$f(n) = O(g(n))$$

# Notação O

- A notação O define um limite superior para a função, por um fator constante
- Escreve-se  $f(n) = O(g(n))$ , se existirem constantes positivas  $c$  e  $n_0$  tais que para  $n \geq n_0$ , o valor de  $f(n)$  é menor ou igual a  $cg(n)$ . Neste caso, pode-se dizer que  $g(n)$  é um limite assintótico superior (em inglês, *asymptotically upper bound*) para  $f(n)$

$$f(n) = O(g(n)), \exists c > 0 \text{ e } n_0 \mid 0 \leq f(n) \leq cg(n), \forall n \geq n_0$$

- Escrevemos  $f(n) = O(g(n))$  para expressar que  $g(n)$  domina assintoticamente  $f(n)$ . Lê-se  $f(n)$  é da ordem no máximo  $g(n)$

# Notação O: Exemplos

- Seja  $f(n) = (n + 1)^2$ 
  - Logo  $f(n)$  é  $O(n^2)$ , quando  $n_0 = 1$  e  $c = 4$ , já que
$$(n + 1)^2 \leq 4n^2 \text{ para } n \geq 1$$
- Seja  $f(n) = n$  e  $g(n) = n^2$ . Mostre que  $g(n)$  não é  $O(n)$ 
  - Sabemos que  $f(n)$  é  $O(n^2)$ , pois para  $n \geq 0$ ,  $n \leq n^2$
  - Suponha que existam constantes  $c$  e  $n_0$  tais que para todo  $n \geq n_0$ ,  $n^2 \leq cn$ . Assim,  $c \geq n$  para qualquer  $n \geq n_0$ . No entanto, não existe uma constante  $c$  que possa ser maior ou igual a  $n$  para todo  $n$

# Notação O: Exemplos

- Mostre que  $g(n) = 3n^3 + 2n^2 + n$  é  $O(n^3)$ 
  - Basta mostrar que  $3n^3 + 2n^2 + n \leq 6n^3$ , para  $n \geq 0$
  - A função  $g(n) = 3n^3 + 2n^2 + n$  é também  $O(n^4)$ , entretanto esta afirmação é mais fraca que dizer que  $g(n)$  é  $O(n^3)$
- Mostre que  $h(n) = \log_5 n$  é  $O(\log n)$ 
  - O  $\log_b n$  difere do  $\log_c n$  por uma constante que no caso é  $\log_b c$
  - Como  $n = c^{\log_c n}$ , tomando o logaritmo base  $b$  em ambos os lados da igualdade, temos que  $\log_b n = \log_b c^{\log_c n} = \log_c n \times \log_b c$

# Notação O

- Quando a notação O é usada para expressar o tempo de execução de um algoritmo no pior caso, está se definindo também o limite superior do tempo de execução desse algoritmo para todas as entradas
- Por exemplo, o algoritmo de ordenação por inserção é  $O(n^2)$  no pior caso
  - Este limite se aplica para qualquer entrada
- O que se quer dizer quando se fala que “*o tempo de execução é  $O(n^2)$* ” é que no pior caso o tempo de execução é  $O(n^2)$ 
  - ou seja, não importa como os dados de entrada estão arranjados, o tempo de execução em qualquer entrada é  $O(n^2)$



# Operações com a notação O

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

$$c \times O(f(n)) = O(f(n)) \quad c = \text{constante}$$

$$O(f(n)) + O(f(n)) = O(f(n))$$

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

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

$$O(f(n))O(g(n)) = O(f(n)g(n))$$

$$f(n)O(g(n)) = O(f(n)g(n))$$

# Operações com a notação O: Exemplos

- Regra da soma  $O(f(n)) + O(g(n))$ 
  - Suponha três trechos cujos tempos de execução sejam  $O(n)$ ,  $O(n^2)$  e  $O(n \log n)$
  - O tempo de execução dos dois primeiros trechos é  $O(\max(n, n^2))$ , que é  $O(n^2)$
  - O tempo de execução de todos os três trechos é então  $O(\max(n^2, \log n))$ , que é  $O(n^2)$
- O produto de  $[\log n + k + O(1/n)]$  por  $[n + O(\sqrt{n})]$  é
$$n \log n + kn + O(\sqrt{n} \log n)$$

# Exercício

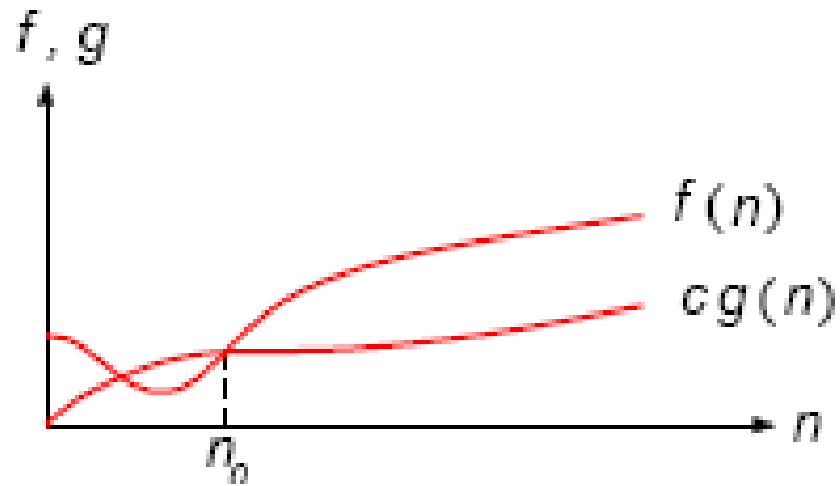
6. Mostre se  $f(n) = O(g(n))$  para os seguintes casos.

a)  $f(n) = \frac{1}{2}n^2 - 3n$  e  $g(n) = n^2$

b)  $f(n) = n \log n - 3n$  e  $g(n) = n^2$

c)  $f(n) = n \log n - 3n$  e  $g(n) = n$

# Notação $\Omega$



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

# Notação $\Omega$

- A notação  $\Omega$  define um limite inferior para a função, por um fator constante
- Escreve-se  $f(n) = \Omega(g(n))$ , se existirem constantes positivas  $c$  e  $n_0$  tais que para  $n \geq n_0$ , o valor de  $f(n)$  é maior ou igual a  $cg(n)$ 
  - Pode-se dizer que  $g(n)$  é um limite assintótico inferior (em inglês, *asymptotically lower bound*) para  $f(n)$

$$f(n) = \Omega(g(n)), \exists c > 0 \text{ e } n_0 \mid 0 \leq cg(n) \leq f(n), \forall n \geq n_0$$

# Notação $\Omega$

- Quando a notação  $\Omega$  é usada para expressar o tempo de execução de um algoritmo no melhor caso, está se definindo também o limite (inferior) do tempo de execução desse algoritmo para todas as entradas
- Por exemplo, o algoritmo de ordenação por inserção é  $\Omega(n)$  no melhor caso
  - O tempo de execução do algoritmo de ordenação por inserção é  $\Omega(n)$
- O que significa dizer que “o tempo de execução” (i.e., sem especificar se é para o pior caso, melhor caso, ou caso médio) é  $\Omega(g(n))$ ?
  - O tempo de execução desse algoritmo é pelo menos uma constante vezes  $g(n)$  para valores suficientemente grandes de  $n$

# Notação $\Omega$ : Exemplos

- Para mostrar que  $f(n) = 3n^3 + 2n^2$  é  $\Omega(n^3)$  basta fazer  $c = 1$ , e então  $3n^3 + 2n^2 \geq n^3$  para  $n \geq 0$

# Limites do algoritmo de ordenação por inserção

- O tempo de execução do algoritmo de ordenação por inserção está entre  $\Omega(n)$  e  $O(n^2)$
- Estes limites são assintoticamente os mais firmes possíveis
  - Por exemplo, o tempo de execução deste algoritmo não é  $\Omega(n^2)$ , pois o algoritmo executa em tempo  $\Theta(n)$  quando a entrada já está ordenada



# Teorema

- Para quaisquer funções  $f(n)$  e  $g(n)$ ,

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

se e somente se,

$$f(n) = O(g(n)), \text{ e}$$

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

# Mais sobre notação assintótica de funções

- Existem duas outras notações na análise assintótica de funções:
  - Notação  $o$  (“O” pequeno)
  - Notação  $\omega$
- Estas duas notações não são usadas normalmente, mas é importante saber seus conceitos e diferenças em relação às notações  $O$  e  $\Omega$ , respectivamente

# Notação o

- O limite assintótico superior definido pela notação  $O$  pode ser assintoticamente firme ou não
  - Por exemplo, o limite  $2n^2 = O(n^2)$  é assintoticamente firme, mas o limite  $2n = O(n^2)$  não é
- A notação  $o$  é usada para definir um limite superior que não é assintoticamente firme
- Formalmente a notação  $o$  é definida como:
$$f(n) = o(g(n)), \text{ para qualquer } c > 0 \text{ e } n_0 \mid 0 \leq f(n) < cg(n), \forall n \geq n_0$$
- Exemplo,  $2n = o(n^2)$  mas  $2n^2 \neq o(n^2)$

# Notação o

- As definições das notações  $O$  e  $o$  são similares
  - A diferença principal é que em  $f(n) = o(g(n))$ , a expressão  $0 \leq f(n) < cg(n)$  é válida para todas constantes  $c > 0$
- Intuitivamente, a função  $f(n)$  tem um crescimento muito menor que  $g(n)$  quando  $n$  tende para infinito. Isto pode ser expresso da seguinte forma:

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

- Alguns autores usam este limite como a definição de  $o$

# Notação $\omega$

- Por analogia, a notação  $\omega$  está relacionada com a notação  $\Omega$  da mesma forma que a notação  $o$  está relacionada com a notação  $O$

- Formalmente a notação  $\omega$  é definida como:

$$f(n) = \omega(g(n)), \text{ para qualquer } c > 0 \text{ e } n_0 \mid 0 \leq cg(n) < f(n), \forall n \geq n_0$$

- Por exemplo,  $\frac{n^2}{2} = \omega(n)$  , mas  $\frac{n^2}{2} \neq \omega(n^2)$

- A relação  $f(n) = \omega(g(n))$  implica em

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

se o limite existir

# Exercício

7. Utilizando as definições para as notações assintóticas, prove se são verdadeiras ou falsas as seguintes afirmativas.

a)  $3n^3 + 2n^2 + n + 1 = O(n^3)$

b)  $7n^2 = O(n)$

c)  $2^{n+2} = O(2^n)$

d)  $2^{2n} = O(2^n)$

e)  $5n^2 + 7n = \Theta(n^2)$

f)  $6n^3 + 5n^2 \neq \Theta(n^2)$

g)  $9n^3 + 3n = \Omega(n)$

h)  $9n^3 + 3n = o(n^3)$

# Comparação de programas

- Podemos avaliar programas comparando as funções de complexidade, negligenciando as constantes de proporcionalidade
- Um programa com tempo de execução  $\Theta(n)$  é melhor que outro com tempo  $\Theta(n^2)$ 
  - Porém, as constantes de proporcionalidade podem alterar esta consideração
- Exemplo: um programa leva  $100n$  unidades de tempo para ser executado e outro leva  $2n^2$ . Qual dos dois programas é melhor?
  - Depende do tamanho do problema
  - Para  $n < 50$ , o programa com tempo  $2n^2$  é melhor do que o que possui tempo  $100n$
  - Para problemas com entrada de dados pequena é preferível usar o programa cujo tempo de execução é  $\Theta(n^2)$
  - Entretanto, quando  $n$  cresce, o programa com tempo de execução  $\Theta(n^2)$  leva muito mais tempo que o programa  $\Theta(n)$

# Classes de Comportamento Assintótico

## Complexidade Constante

- $f(n) = \Theta(1)$ 
  - O uso do algoritmo independe do tamanho de  $n$
  - As instruções do algoritmo são executadas um número fixo de vezes
  - O que significa um algoritmo ser  $\Theta(2)$  ou  $\Theta(5)$ ?



# Classes de Comportamento Assintótico

## Complexidade logarítmica

- $f(n) = \Theta(\log n)$ 
  - Ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores
  - Nestes casos, o tempo de execução pode ser considerado como sendo menor do que uma constante grande
- Supondo que a base do logaritmo seja 2:
  - Para  $n = 1\,000$ ,  $\log_2 \approx 10$
  - Para  $n = 1\,000\,000$ ,  $\log_2 \approx 20$
- Exemplo:
  - Algoritmo de pesquisa binária

# Classes de Comportamento Assintótico

## Complexidade linear

- $f(n) = \Theta(n)$ 
  - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada
  - Esta é a melhor situação possível para um algoritmo que tem que processar/produzir  $n$  elementos de entrada/saída
  - Cada vez que  $n$  dobra de tamanho, o tempo de execução também dobra
- Exemplo:
  - Algoritmo de pesquisa sequencial

# Classes de Comportamento Assintótico

## Complexidade linear logarítmica

- $f(n) = \Theta(n \log n)$ 
  - Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois agrupando as soluções
  - Caso típico dos algoritmos baseados no paradigma **divisão-e-conquista**
- Supondo que a base do logaritmo seja 2:
  - Para  $n = 1\,000\,000$ ,  $\log_2 \approx 20\,000\,000$
  - Para  $n = 2\,000\,000$ ,  $\log_2 \approx 42\,000\,000$
- Exemplo:
  - Algoritmo de ordenação MergeSort

# Classes de Comportamento Assintótico

## Complexidade quadrática

- $f(n) = \Theta(n^2)$ 
  - Algoritmos desta ordem de complexidade ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro do outro
  - Para  $n = 1000$ , o número de operações é da ordem de 1000000
  - Sempre que  $n$  dobra o tempo de execução é multiplicado por 4
  - Algoritmos deste tipo são úteis para resolver problemas de tamanhos relativamente pequenos
- Exemplos:
  - Algoritmos de ordenação simples como seleção e inserção

# Classes de Comportamento Assintótico

## Complexidade cúbica

- $f(n) = \Theta(n^3)$ 
  - Algoritmos desta ordem de complexidade geralmente são úteis apenas para resolver problemas relativamente pequenos
  - Para  $n = 100$ , o número de operações é da ordem de 1000000
  - Sempre que  $n$  dobra o tempo de execução é multiplicado por 8
- Exemplo:
  - Algoritmo para multiplicação de matrizes

# Classes de Comportamento Assintótico

## Complexidade Exponencial

- $f(n) = \Theta(2^n)$ 
  - Algoritmos desta ordem de complexidade não são úteis sob o ponto de vista prático
  - Eles ocorrem na solução de problemas quando se usa a força bruta para resolvê-los
  - Para  $n = 20$ , o tempo de execução é cerca de 1000000
  - Sempre que  $n$  dobra o tempo de execução fica elevado ao quadrado
- Exemplo:
  - Algoritmo do Caixeiro Viajante

# Classes de Comportamento Assintótico

## Complexidade Exponencial

- $f(n) = \Theta(n!)$ 
  - Um algoritmo de complexidade  $\Theta(n!)$  é dito ter complexidade exponencial, apesar de  $\Theta(n!)$  ter comportamento muito pior do que  $\Theta(2^n)$
  - Geralmente ocorrem quando se usa força bruta na solução do problema
- Considerando:
  - $n = 20$ , temos que  $20! = 2432902008176640000$ , um número com 19 dígitos
  - $n = 40$  temos um número com 48 dígitos

# Comparação de funções de complexidade

Função de custo	Tamanho $n$					
	10	20	30	40	50	60
$n$	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
$n^2$	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0.35 s	0,0036 s
$n^3$	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0.316 s
$n^5$	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
$2^n$	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
$3^n$	0,059 s	58 min	6,5 anos	3855 séc.	$10^8$ séc.	$10^{13}$ séc.

Função de custo de tempo	Computador atual	Computador 100 vezes mais rápido	Computador 1000 vezes mais rápido
$n$	$t_1$	$100 t_1$	$1000 t_1$
$n^2$	$t_2$	$10 t_2$	$31,6 t_2$
$n^3$	$t_3$	$4,6 t_3$	$10 t_3$
$2^n$	$t_4$	$t_4 + 6,6$	$t_4 + 10$



# Algoritmo exponencial × Algoritmo polinomial

- Funções de complexidade:
  - Um algoritmo cuja função de complexidade é  $\Omega(c^n)$ ,  $c > 1$ , é chamado de algoritmo exponencial no tempo de execução
  - Um algoritmo cuja função de complexidade é  $O(p(n))$ , onde  $p(n)$  é um polinômio de grau  $n$ , é chamado de algoritmo polinomial no tempo de execução
- A distinção entre estes dois tipos de algoritmos torna-se significativa quando o tamanho do problema a ser resolvido cresce
- Esta é a razão porque algoritmos polinomiais são muito mais úteis na prática do que algoritmos exponenciais
  - Geralmente, algoritmos exponenciais são simples variações de pesquisa exaustiva

# Algoritmo exponencial × Algoritmo polinomial

- Os algoritmos polinomiais são geralmente obtidos através de um entendimento mais profundo da estrutura do problema
- Tratabilidade dos problemas:
  - Um problema é considerado **intratável** se ele é tão difícil que não se conhece um algoritmo polinomial para resolvê-lo
  - Um problema é considerado **tratável** (bem resolvido) se existe um algoritmo polinomial para resolvê-lo
- Aspecto importante no projeto de algoritmos

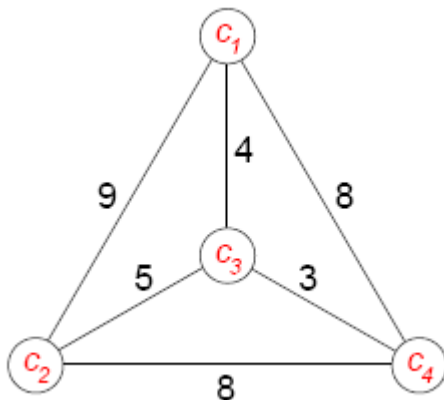
# Algoritmo exponencial × Algoritmo polinomial

- A distinção entre algoritmos polinomiais eficientes e algoritmos exponenciais ineficientes possui várias exceções
- Exemplo: um algoritmo com função de complexidade  $f(n) = 2^n$  é mais rápido que um algoritmo  $g(n) = n^5$  para valores de  $n$  menores ou iguais a 20
- Também existem algoritmos exponenciais que são muito úteis na prática
  - Exemplo: o algoritmo Simplex para programação linear possui complexidade de tempo exponencial para o pior caso mas executa muito rápido na prática.
- Tais exemplos não ocorrem com frequência na prática, e muitos algoritmos exponenciais conhecidos não são muito úteis.

# Algoritmo exponencial

## O Problema do Caixeiro Viajante

- Um **caixeiro viajante** deseja visitar  $n$  cidades de tal forma que sua viagem inicie e termine em uma mesma cidade, e cada cidade deve ser visitada uma única vez
- Supondo que sempre há uma estrada entre duas cidades quaisquer, o problema é encontrar a menor rota para a viagem
- Seja a figura que ilustra o exemplo para quatro cidades  $c_1$ ,  $c_2$ ,  $c_3$  e  $c_4$  em que os números nas arestas indicam a distância entre duas cidades



O percurso  $\langle c_1, c_3, c_4, c_2, c_1 \rangle$  é uma solução para o problema, cujo percurso total tem distância 24

# Exemplo de algoritmo exponencial

- Um algoritmo simples seria verificar todas as rotas e escolher a menor delas
- Há  $(n - 1)!$  rotas possíveis e a distância total percorrida em cada rota envolve  $n$  adições, logo o número total de adições é  $n!$
- No exemplo anterior teríamos  $24$  adições
- Suponha agora  $50$  cidades: o número de adições seria  $50! \approx 10^{64}$
- Em um computador que executa  $10^9$  adições por segundo, o tempo total para resolver o problema com  $50$  cidades seria maior do que  $10^{45}$  séculos só para executar as adições
- O problema do caixeiro viajante aparece com frequência em problemas relacionados com transporte, mas também aplicações importantes relacionadas com otimização de caminho percorrido

# Fundamentos Matemáticos

Cormen – págs. 835 até 844

---

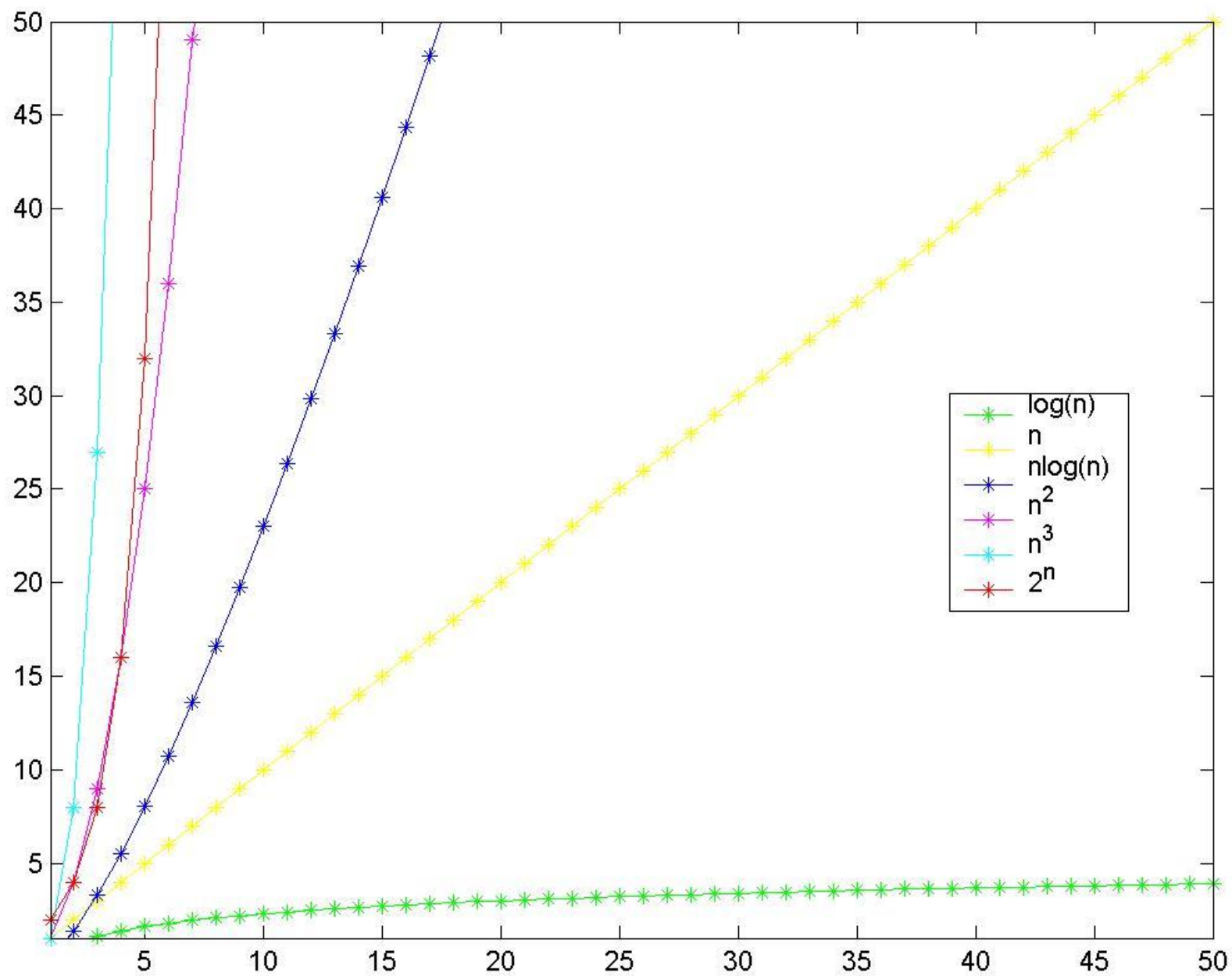
# Hierarquias de funções

- A seguinte hierarquia de funções pode ser definida do ponto de vista assintótico:

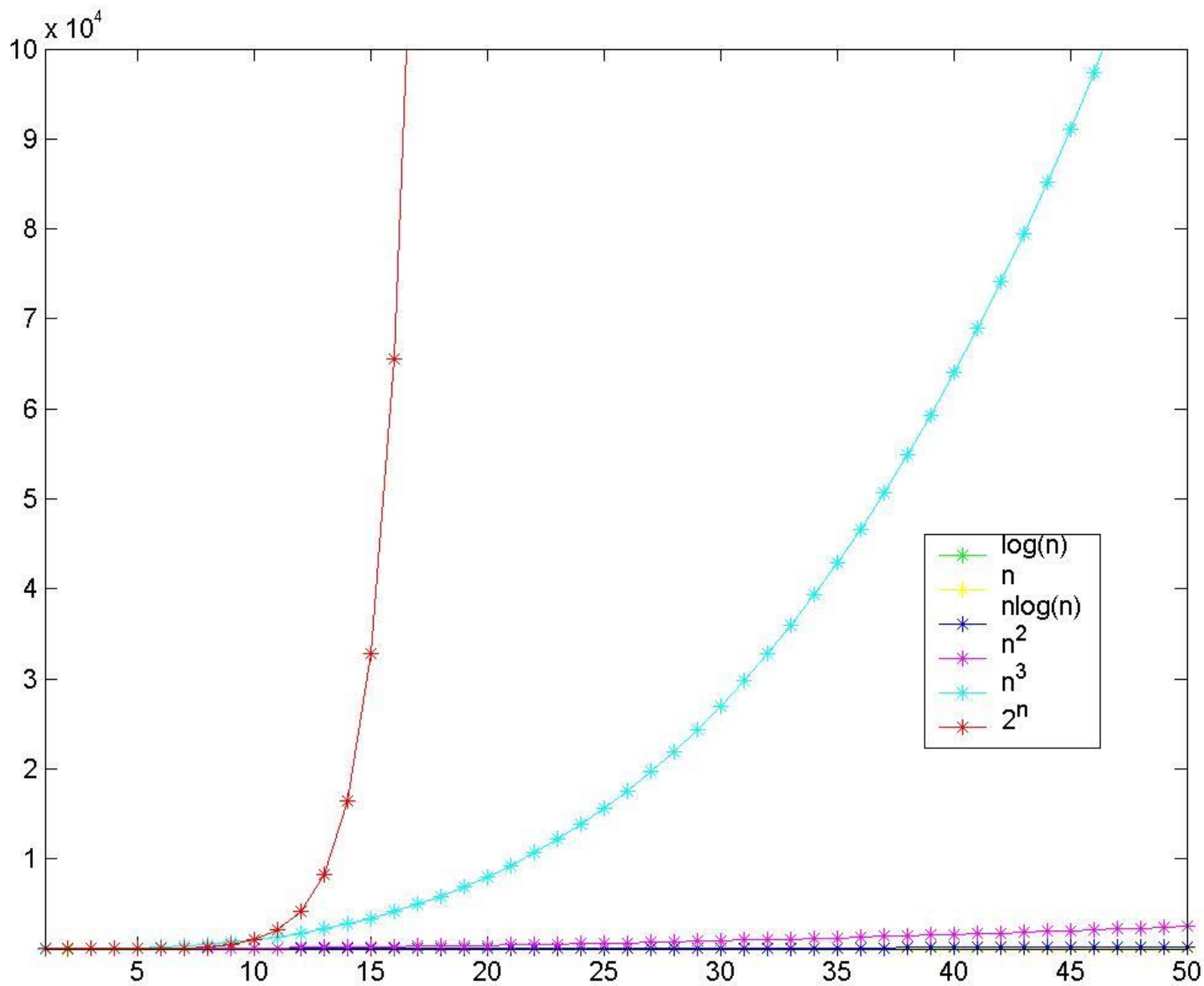
$$1 \prec \log \log n \prec \log n \prec n^\varepsilon \prec n \prec n^c \prec n^{\log n} \prec c^n \prec n^n \prec c^{c^n}$$

onde  $\varepsilon$  e  $c$  são constantes arbitrárias com  $0 < \varepsilon < 1 < c$

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







# Funções Usuais

- Logaritmos e Exponenciais:

$$a^x = y \Leftrightarrow \log_a y = x$$

$$\log_a a^x = x$$

$$a^0 = 1 \implies \log_a 1 = 0$$

$$a^{x+y} = a^x \times a^y \implies \log_a p + \log_a q = \log_a pq$$

$$a^{x-y} = \frac{a^x}{a^y} \implies \log_a \frac{p}{q} = \log_a p - \log_a q$$

$$(a^x)^y = a^{xy} \implies \log_a x^y = y \log_a x$$

$$(a^x)^y = a^{xy} \implies \log_a x = \frac{\log_b x}{\log_b a}$$

- Aproximação de Stirling:

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

# Somatórios

- Notação de somatório:  $\sum_{i=1}^n a_i = a_1 + a_2 + \cdots + a_n$
- Propriedades:  $\sum_{i=1}^n (ca_i + b_i) = c \sum_{i=1}^n a_i + \sum_{i=1}^n b_i$

# Alguns somatórios

$$\sum_{i=1}^n 1 = n$$

- Série aritmética

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

- Somas de quadrados e cubos

$$\sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=0}^n i^3 = \frac{n^2(n+1)^2}{4}$$

# Alguns somatórios

- Série geométrica (ou exponencial)

- Para  $a \neq 1$

$$\sum_{i=0}^n a^i = 1 + a + a^2 + \cdots + a^n$$

$$\sum_{i=0}^n a^i = \frac{a^{n+1} - 1}{a - 1}$$

- Para  $|a| < 1$

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1 - a}$$

# Integração e diferenciação de séries

- Fórmulas adicionais podem ser obtidas por integração ou diferenciação das fórmulas anteriores
  - Exemplo: diferenciando-se ambos os lados de:

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}$$

- temos:

$$\sum_{i=0}^{\infty} i a^i = \frac{a}{(1-a)^2}$$

# Técnicas de Análise de Algoritmos

Ziviani – págs. 19 até 23 e 35 até 42

Cormen – págs. 21 até 31 e 50 até 72

# Técnicas de análise de algoritmos

- Determinar o tempo de execução de um programa pode ser um problema matemático complexo
- Determinar a ordem do tempo de execução, sem preocupação com o valor das constantes envolvidas, pode ser uma tarefa mais simples
- A análise utiliza técnicas de matemática discreta, envolvendo contagem ou enumeração dos elementos de um conjunto:
  - manipulação de somas
  - produtos
  - permutações
  - fatoriais
  - coeficientes binomiais
  - solução de **equações de recorrência**



# Análise do tempo de execução

- Comando de atribuição, de leitura ou de escrita:  $\Theta(1)$
- Sequência de comandos: determinado pelo maior tempo de execução de qualquer comando da sequência
- Comando de decisão: tempo dos comandos dentro do comando condicional, mais tempo para avaliar a condição, que é  $\Theta(1)$
- Anel: soma do tempo de execução do corpo do anel mais o tempo de avaliar a condição para terminação (geralmente  $\Theta(1)$ ), multiplicado pelo número de iterações

# Análise do tempo de execução

- Procedimentos não recursivos:
  - Cada um deve ser computado separadamente um a um, iniciando com os que não chamam outros procedimentos
  - Avalia-se então os que chamam os já avaliados (utilizando os tempos desses)
  - O processo é repetido até chegar no programa principal
- Procedimentos recursivos:
  - É associada uma função de complexidade  $T(n)$  desconhecida, onde  $n$  mede o tamanho dos argumentos
  - Obtemos uma equação de recorrência para  $T(n)$
  - Resolvemos a equação de recorrência

# Análise de algoritmos não recursivos

- Considerando que a operação relevante seja o número de atribuições à variável *a*, qual é a **função de complexidade** da função exemplo1?
- Qual a **ordem de complexidade** da função exemplo1?

```
void exemplo1 (int n)
{
    int i, a;
    a=0;
    for (i=0; i<n; i++)
        a+=i;
}
```

# Análise de algoritmos não recursivos

- Considerando que a operação relevante seja o número de atribuições à variável *a*, qual é a **função de complexidade** da função exemplo2?
- Qual a **ordem de complexidade** da função exemplo2?

```
void exemplo2 (int n)
{
    int i,j,a;
    a=0;
    for (i=0; i<n; i++)
        for (j=n; j>i; j--)
            a+=i+j;
    exemplo1(n);
}
```

# Análise de algoritmos não recursivos

- Ordenação por Seleção
  - Seleciona o menor elemento do conjunto
  - Troca este com o primeiro elemento  $A[0]$
  - Repita as duas operações acima com os  $n - 1$  elementos restantes, depois com os  $n - 2$ , até que reste apenas um

# Análise de algoritmos não recursivos

```
void Ordena (int A[]) {  
    /*ordena o vetor A em ordem ascendente*/  
    int i, j, min, x;  
(1)   for (i = 0; i < n-1; i++) {  
(2)       min = i;  
(3)       for (j = i + 1; j < n; j++)  
(4)           if (A[j] < A[min])  
(5)               min = j;  
        /*troca A[min] e A[i]*/  
(6)       x = A[min];  
(7)       A[min] = A[i];  
(8)       A[i] = x;  
    }  
}
```

# Análise de algoritmos não recursivos

- Considerando que a operação relevante seja o número de comparações com os elementos do vetor:

$$\begin{aligned}T(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i-1+1) = \sum_{i=0}^{n-2} (n-i-1) \\&= \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1 = n(n-1) - \frac{(n-2)(n-1)}{2} - (n-1) \\&= n^2 - n - \frac{n^2 - 3n + 2}{2} - n + 1 = \frac{n^2}{2} - \frac{n}{2}\end{aligned}$$

- Se considerarmos o número de movimentações com os elementos de  $A$ , o programa realiza exatamente  $3(n-1)$  operações

# Exercício

8. O que faz essa função ? Qual é a operação relevante? Qual a sua ordem de complexidade?

```
void p1 (int n)
{
    int i, j, k;

    for (i=0; i<n; i++)
        for (j=0; j<n; j++) {
            C[i][j]=0;
            for (k=n-1; k>=0; k--)
                C[i][j]=C[i][j]+A[i][k]*B[k][j];
        }
}
```



# Exercício

9. O que faz essa função ? Qual é a operação relevante? Qual a sua ordem de complexidade?

```
void p2 (int n)
{
    int i, j, x, y;

    x = y = 0;
    for (i=1; i<=n; i++) {
        for (j=i; j<=n; j++)
            x = x + 1;
        for (j=1; j<i; j++)
            y = y + 1;
    }
}
```

# Exercício

10. Qual é a função de complexidade para o número de atribuições ao vetor  $x$ ?

```
void Exercicio3(int n){
    int i, j, a;

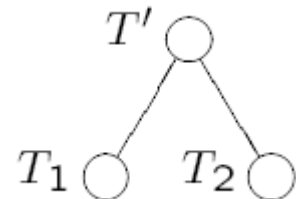
    for (i=0; i<n; i++){
        if (x[i] > 10)
            for (j=i+1; j<n; j++)
                x[j] = x[j] + 2;
        else {
            x[i] = 1;
            j = n-1;
            while (j >= 0) {
                x[j] = x[j] - 2;
                j = j - 1;
            }
        }
    }
}
```

# Algoritmos recursivos

- Um objeto é recursivo quando é definido parcialmente em termos de si mesmo
- Um algoritmo que chama a si mesmo, direta ou indiretamente, é dito ser **recursivo**
- Recursividade permite descrever algoritmos de forma mais clara e concisa, especialmente problemas recursivos por natureza ou que utilizam estruturas recursivas

# Algoritmos recursivos

- Exemplo 1: Números naturais
  - a) 1 é um número natural
  - b) O sucessor de um número natural é um número natural
- Exemplo 2: Função fatorial
  - a)  $0! = 1$
  - b) Se  $n > 0$  então  $n! = n (n - 1)!$
- Exemplo 3: Árvores
  - a) A árvore vazia é uma árvore
  - b) Se  $T_1$  e  $T_2$  são árvores então  $T'$  é uma árvore



# Algoritmos recursivos

- Normalmente, as funções recursivas são divididas em duas partes
  - Chamada recursiva
  - Condição de parada
- A chamada recursiva pode ser direta (mais comum) ou indireta (A chama B que chama A novamente)
- A condição de parada é fundamental para evitar a execução de *loops* infinitos

# Algoritmos recursivos

- Internamente, quando qualquer chamada de função é feita dentro de um programa, é criado um registro de ativação na pilha de execução do programa
- O registro de ativação armazena os parâmetros e variáveis locais da função bem como o “ponto de retorno” no programa ou subprograma que chamou essa função
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função

# Exemplo: fatorial recursivo

```
int fat (int n) {  
    if (n<=0)  
        return (1);  
    else  
        return (n * fat(n-1));  
}
```

```
int main() {  
    int f;  
    f = fat(5);  
    printf("%d",f);  
    return (0);  
}
```

$\text{fat}(5) = 5 * \text{fat}(4)$

$\text{fat}(4) = 4 * \text{fat}(3)$

$\text{fat}(3) = 3 * \text{fat}(2)$

$\text{fat}(2) = 2 * \text{fat}(1)$

$\text{fat}(1) = 1 * \text{fat}(0)$

$\text{fat}(0) = 1$

# Complexidade: fatorial recursivo

- A complexidade de tempo do fatorial recursivo é  $O(n)$  (em breve iremos ver a maneira de calcular isso usando **equações de recorrência**)
- Mas a complexidade de espaço também é  $O(n)$ , devido a pilha de execução



# Complexidade: fatorial iterativo

- A complexidade de espaço do fatorial não recursivo é  $O(1)$

```
int fatiter (int n) {  
    int f;  
    f = 1;  
    while(n > 0) {  
        f = f * n;  
        n = n - 1;  
    }  
    return (f);  
}
```

# Recursividade

- Portanto, a recursividade nem sempre é a melhor solução, mesmo quando a definição matemática do problema é feita em termos recursivos

# Exemplo: série de Fibonacci

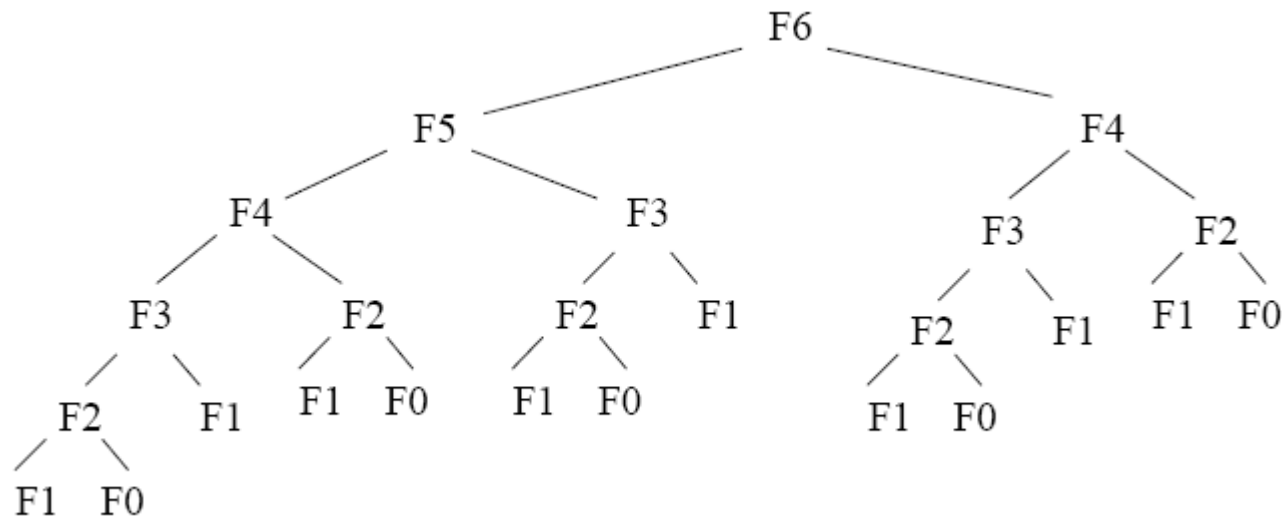
- Série de Fibonacci:
  - $F_n = F_{n-1} + F_{n-2}$        $n > 2$ ,
  - $F_0 = F_1 = 1$
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

```
int Fib(int n) {  
    if (n<2)  
        return (1);  
    else  
        return (Fib(n-1) + Fib(n-2));  
}
```

# Complexidade: série de Fibonacci

- Ineficiência em Fibonacci

- Termos  $F_{n-1}$  e  $F_{n-2}$  são computados independentemente
- Número de chamadas recursivas = número de Fibonacci!
- Custo para cálculo de  $F_n$ 
  - ▶  $O(\phi^n)$  onde  $\phi = (1 + \sqrt{5})/2 = 1,61803...$
  - ▶ Exponencial!!!



# Exemplo: série de Fibonacci iterativo

```
int FibIter(int n) {  
    int i, k, F;  
  
    i = 1; F = 0;  
    for (k = 1; k <= n; k++) {  
        F += i;  
        i = F - i;  
    }  
    return F;  
}
```

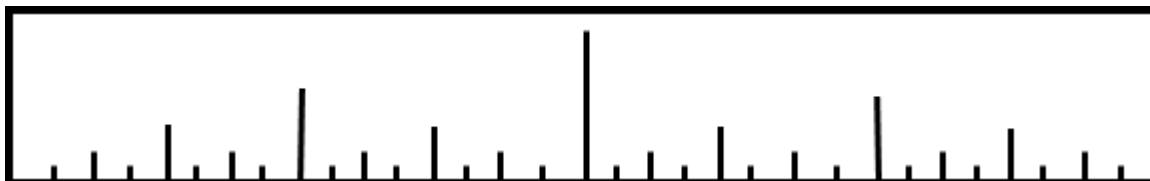
- Complexidade:  $O(n)$
- Conclusão: não usar recursividade cegamente!

# Quando vale a pena usar recursividade?

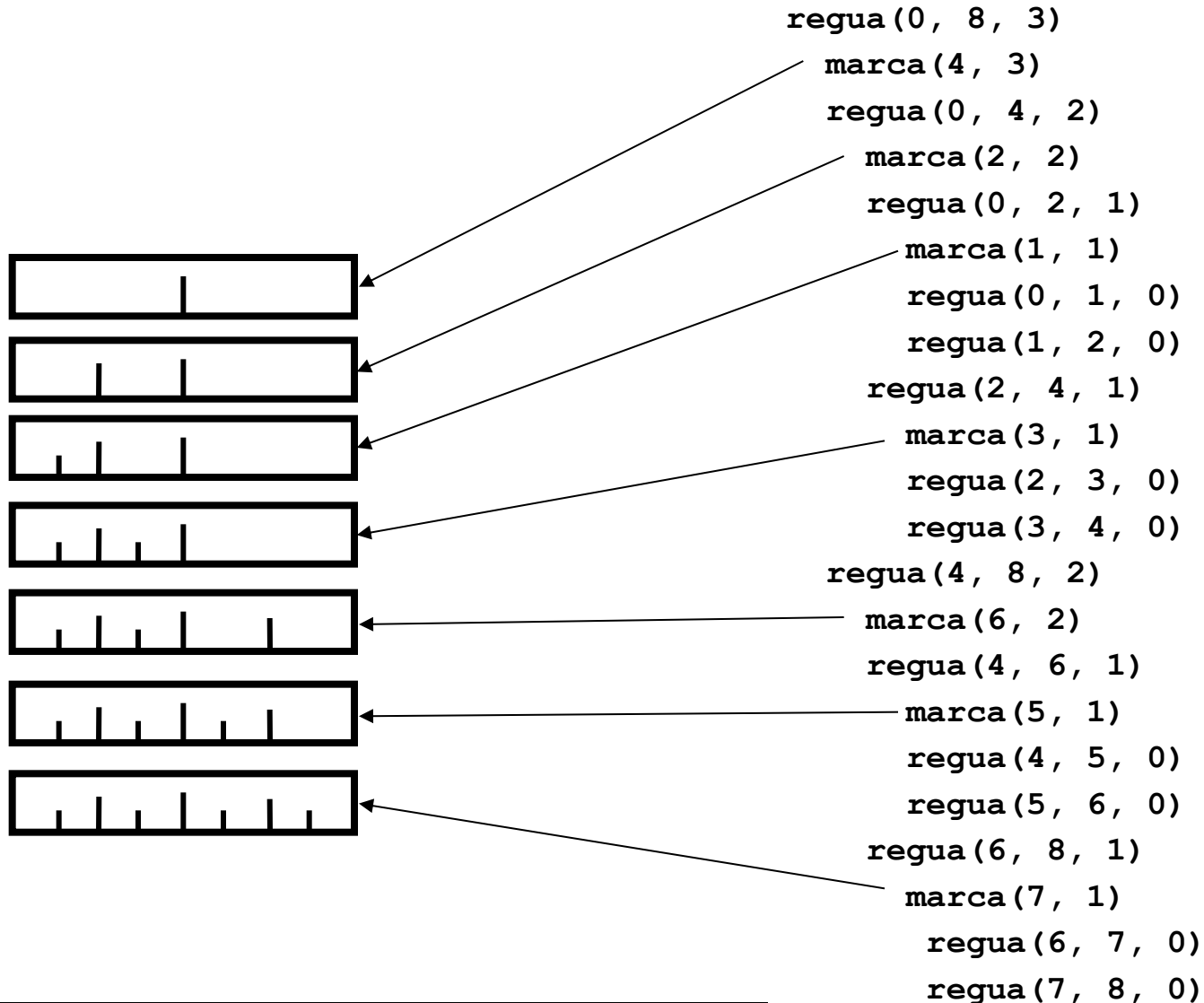
- Recursividade vale a pena para algoritmos complexos, cuja a implementação iterativa é complexa e normalmente requer o uso explícito de uma pilha
  - Dividir para Conquistar (Ex. Quicksort)
  - Caminhamento em Árvores (pesquisa, backtracking)
- Evitar o uso de recursividade quando existe uma solução óbvia por iteração:
  - Fatorial
  - Série de Fibonacci

# Exemplo: régua

```
void regua(int l, r, h){  
    int m;  
  
    if (h > 0) {  
        m = (l + r) / 2;  
        marca(m, h);  
        regua(l, m, h - 1);  
        regua(m, r, h - 1);  
    }  
}
```

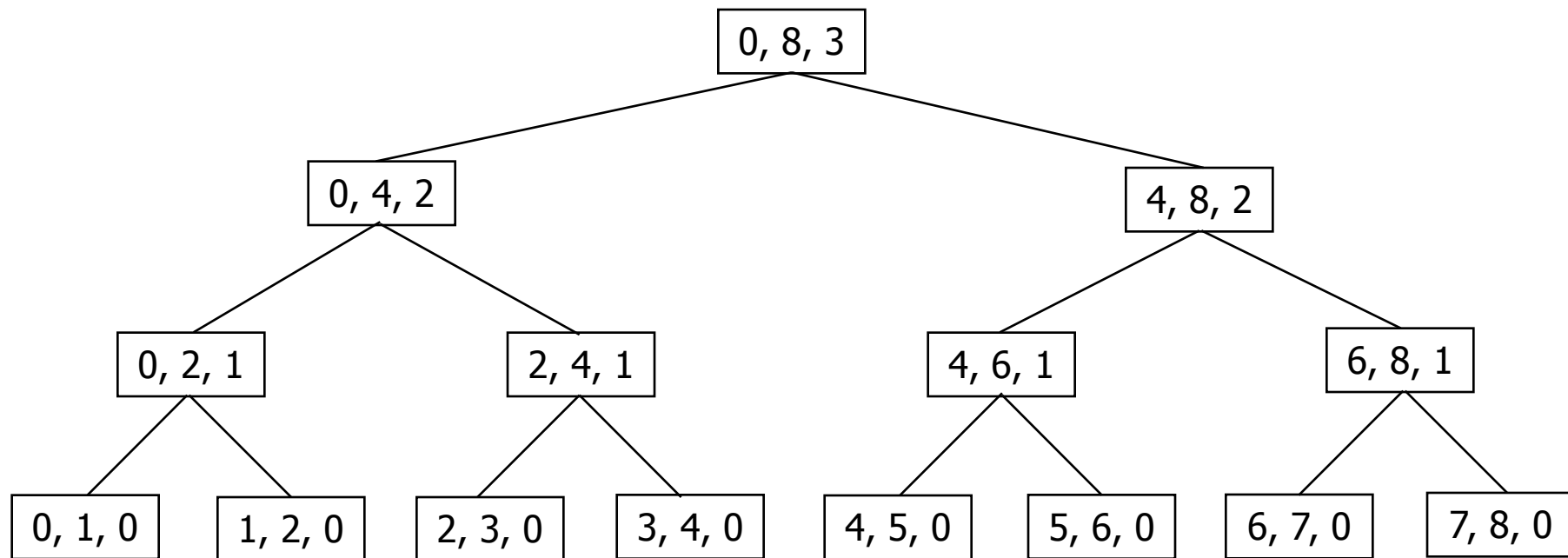


# Exemplo: régua





# Exemplo: régua



# Exercícios

11. Implemente uma função recursiva para computar o valor de  $2^n$

12. O que faz a função abaixo?

```
int f(int a, int b) {  
    // considere a > b  
    if (b == 0)  
        return a;  
    else  
        return (f(b, a%b));  
}
```

# Análise de procedimento recursivo

```
Pesquisa(n);  
(1) if  $n \leq 1$   
(2) then "inspecione elemento" e termine  
   else begin  
(3)       para cada um dos  $n$  elementos "inspecione elemento";  
(4)       Pesquisa( $n-1$ );  
   end;
```

- Para cada procedimento recursivo é associada uma função de complexidade  $T(n)$  desconhecida, onde  $n$  mede o tamanho dos argumentos para o procedimento
- Obtemos uma equação de recorrência para  $T(n)$
- **Equação de recorrência:** maneira de definir uma função por uma expressão envolvendo a mesma função

# Análise de procedimento recursivo

- Seja  $T(n)$  uma função de complexidade que represente o número de inspeções nos  $n$  elementos do conjunto.
- O custo de execução das linhas (1) e (2) é  $O(1)$  e da linha (3) é  $O(n)$
- Usa-se uma **equação de recorrência** para determinar o número de chamadas recursivas
- O termo  $T(n)$  é especificado em função dos termos anteriores  $T(1), T(2), \dots, T(n-1)$

$$\begin{cases} T(n) = T(n-1) + n \\ T(1) = 1 \end{cases} \quad (\text{para } n = 1, \text{ fazemos uma inspeção})$$

# Resolução de equação de recorrência

$$\begin{cases} T(n) = T(n-1) + n \\ T(1) = 1 \end{cases}$$

$$\left. \begin{array}{l} T(n) = T(\cancel{n}-1) + n \\ T(\cancel{n}-1) = T(\cancel{n}-2) + n-1 \\ T(\cancel{n}-2) = T(\cancel{n}-3) + n-2 \\ T(\cancel{n}-3) = T(\cancel{n}-4) + n-3 \\ \quad \quad \quad \vdots \\ T(n - (\cancel{n}-2)) = T(n - \cancel{(n-1)}) + 2 \\ T(n - \cancel{(n-1)}) = 1 \end{array} \right\} n + (n-1) + (n-2) + \cdots + 2 + 1 = \sum_{i=1}^n i$$

---

$$T(n) = \sum_{i=1}^n i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

Logo, o programa do exemplo é  $\Theta(n^2)$

# Análise da função `fat`

- Seja a seguinte função para calcular o fatorial de  $n$ :

```
int fat (int n) {  
    if (n<=1)  
        return (1);  
    else  
        return (n * fat(n-1));  
}
```

- Seja a seguinte equação de recorrência para esta função:

$$\begin{cases} T(n) = T(n-1) + c & n > 1 \\ T(1) = d \end{cases}$$

- Esta equação diz que quando  $n = 1$ , o custo para executar `fat` é igual a  $d$
- Para valores de  $n$  maiores que 1, o custo para executar `fat` é  $c$  mais o custo para executar  $T(n-1)$

# Resolvendo a equação de recorrência

$$\begin{cases} T(n) = T(n-1) + c & n > 1 \\ T(1) = d \end{cases}$$

$$\begin{array}{l} T(n) = T(\cancel{n}-1) + c \\ T(\cancel{n}-1) = T(\cancel{n}-2) + c \\ T(\cancel{n}-2) = T(\cancel{n}-3) + c \\ \quad \quad \quad \vdots \\ T(n - (\cancel{n}-2)) = T(\cancel{n} - (\cancel{n}-1)) + c \\ T(n - (\cancel{n}-1)) = d \end{array} \left. \vphantom{\begin{array}{l} T(n) = T(\cancel{n}-1) + c \\ T(\cancel{n}-1) = T(\cancel{n}-2) + c \\ T(\cancel{n}-2) = T(\cancel{n}-3) + c \\ \quad \quad \quad \vdots \\ T(n - (\cancel{n}-2)) = T(\cancel{n} - (\cancel{n}-1)) + c \\ T(n - (\cancel{n}-1)) = d \end{array}} \right\} \sum_{i=1}^{n-1} c + d$$

---

$$T(n) = c(n-1) + d$$

Logo, o programa do exemplo é  $O(n)$

# Exercícios

**13.** Resolva as seguintes equações de recorrência:

$$\text{a) } \begin{cases} T(n) = T\left(\frac{n}{2}\right) + 1 & (n \geq 2) \\ T(1) = 0 & (n = 1) \end{cases}$$

$$\text{b) } \begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + n & (n \geq 2) \\ T(1) = 0 & (n = 1) \end{cases}$$

$$\text{c) } \begin{cases} T(n) = T\left(\frac{n}{3}\right) + n & (n > 1) \\ T(1) = 1 & (n = 1) \end{cases}$$



# Teorema Mestre

- Recorrências da forma

$$T(n) = aT(n/b) + f(n)$$

onde  $a \geq 1$  e  $b > 1$  são constantes e  $f(n)$  é uma função assintoticamente positiva, podem ser resolvidas usando o Teorema Mestre

- Neste caso, não estamos achando a forma fechada da recorrência mas sim seu comportamento assintótico

# Teorema Mestre

- Sejam as constantes  $a \geq 1$  e  $b > 1$  e  $f(n)$  uma função definida nos inteiros não-negativos pela recorrência:

$$T(n) = aT(n/b) + f(n)$$

onde a fração  $n/b$  pode significar  $\lfloor n/b \rfloor$  ou  $\lceil n/b \rceil$ . A equação de recorrência  $T(n)$  pode ser limitada assintoticamente da seguinte forma:

1. Se  $f(n) = O(n^{\log_b a - \varepsilon})$  para alguma constante  $\varepsilon > 0$ , então  $T(n) = \Theta(n^{\log_b a})$
2. Se  $f(n) = \Theta(n^{\log_b a})$ , então  $T(n) = \Theta(n^{\log_b a} \log n)$
3. Se  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  para alguma constante  $\varepsilon > 0$  e se  $af(n/b) \leq cf(n)$  para alguma constante  $c < 1$  e para  $n$  suficientemente grande, então  $T(n) = \Theta(f(n))$

# Comentários sobre o teorema Mestre

- Nos três casos estamos comparando a função  $f(n)$  com a função  $n^{\log_b a}$ . Intuitivamente, a solução da recorrência é determinada pela maior das duas funções.
- Por exemplo:
  - No primeiro caso a função  $n^{\log_b a}$  é a maior e a solução para a recorrência é  $T(n) = \Theta(n^{\log_b a})$
  - No terceiro caso, a função  $f(n)$  é a maior e a solução para a recorrência é  $T(n) = \Theta(f(n))$
  - No segundo caso, as duas funções são do mesmo “tamanho”. Neste caso, a solução fica multiplicada por um fator logarítmico e fica da forma  $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$

# Tecnicidades sobre o teorema Mestre

- No primeiro caso, a função  $f(n)$  deve ser não somente menor que  $n^{\log_b a}$  mas ser polinomialmente menor. Ou seja,  $f(n)$  deve ser assintoticamente menor que  $n^{\log_b a}$  por um fator de  $n^\varepsilon$ , para alguma constante  $\varepsilon > 0$
- No terceiro caso, a função  $f(n)$  deve ser não somente maior que  $n^{\log_b a}$  mas ser polinomialmente maior e satisfazer a condição de “regularidade” que  $af(n/b) \leq cf(n)$ . Esta condição é satisfeita pela maior parte das funções polinomiais encontradas neste curso.

# Tecnicidades sobre o teorema Mestre

- Teorema **não** cobre todas as possibilidades para  $f(n)$ :
  - Entre os casos 1 e 2 existem funções  $f(n)$  que são menores que  $n^{\log_b a}$  mas não são polinomialmente menores
  - Entre os casos 2 e 3 existem funções  $f(n)$  que são maiores que  $n^{\log_b a}$  mas não são polinomialmente maiores

Se a função  $f(n)$  cai numa dessas condições, ou a condição de regularidade do caso 3 é falsa, então não se pode aplicar este teorema para resolver a recorrência

# Uso do teorema: Exemplo 1

$$T(n) = 9T(n/3) + n$$

- Temos que,

$$a = 9, \quad b = 3, \quad f(n) = n$$

- Desta forma,

$$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$$

- Como  $f(n) = O(n^{\log_3 9 - \varepsilon})$ , onde  $\varepsilon = 1$ , podemos aplicar o caso 1 do teorema e concluir que a solução da recorrência é

$$T(n) = \Theta(n^2)$$

# Uso do teorema: Exemplo 2

$$T(n) = T(2n/3) + 1$$

- Temos que,

$$a = 1, \quad b = 3/2, \quad f(n) = 1$$

- Desta forma,

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

- O caso 2 se aplica já que  $f(n) = O(n^{\log_b a}) = \Theta(1)$ . Temos então que a solução da recorrência é

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

# Uso do teorema: Exemplo 3

$$T(n) = 3T(n/4) + n \log n$$

- Temos que,

$$a = 3, \quad b = 4, \quad f(n) = n \log n$$

- Desta forma,

$$n^{\log_b a} = n^{\log_4 3} = O(n^{0,793})$$

- Como  $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$ , onde  $\varepsilon \approx 0,2$ , o caso 3 se aplica se mostrarmos que a condição de regularidade é verdadeira para  $f(n)$



# Uso do teorema: Exemplo 3

- Para um valor suficientemente grande de  $n$

$$af(n/b) = 3(n/4)\log(n/4) \leq (3/4)n\log n = cf(n)$$

- Para  $c = 3/4$ . Consequentemente, usando o caso 3, a solução para a recorrência é:

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

# Uso do teorema: Exemplo 4

$$T(n) = 2T(n/2) + n \log n$$

- Temos que,

$$a = 2, \quad b = 2, \quad f(n) = n \log n$$

- Desta forma,

$$n^{\log_b a} = n$$

- Aparentemente o caso 3 deveria se aplicar já que  $f(n) = n \log n$  é assintoticamente maior que  $n^{\log_b a} = n$ . Mas no entanto, não é polinomialmente maior. A fração  $f(n)/n^{\log_b a} = (n \log n)/n = \log n$  que é assintoticamente menor que  $n^\varepsilon$  para toda constante positiva  $\varepsilon$ . Consequentemente, a recorrência cai na situação entre os casos 2 e 3 onde o teorema não pode ser aplicado.

# Exercício

14. Use o teorema mestre para derivar um limite assintótico  $\Theta$  para da seguinte recorrência:

$$T(n) = 4T(n/2) + n$$