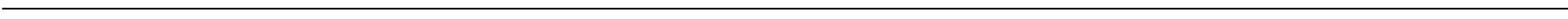


**Pontifícia Universidade Católica de Minas Gerais**  
**Instituto de Ciências Exatas e Informática**  
**Departamento de Ciência da Computação**  
**Curso de Ciência da Computação**

# **Projeto e Análise de Algoritmos**

## **Parte 1**

**Raquel Mini**  
**[raquelmini@pucminas.br](mailto: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

		$2^n$ na máquina $10^4$
	<b>tempo (s)</b>	<b>tamanho</b>
	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

	$2^n$ na máquina $10^4$	$2^n$ na máquina $10^9$
tempo (s)	tamanho	tamanho
0,10	10	27
1	13	30
1 minuto	60	19
1 hora	3.600	25
1 dia	86.400	30
1 ano	31.536.000	38

*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

	$2^n$ na máquina $10^4$	$2^n$ na máquina $10^9$	$n^2$ na máquina $10^4$	$n^2$ na máquina $10^9$
tempo (s)	tamanho	tamanho	tamanho	tamanho
0,10	10	27	32	10.000
1	13	30	100	31.623
1 minuto	60	36	775	244.949
1 hora	3.600	42	6.000	1.897.367
1 dia	86.400	46	29.394	9.295.160
1 ano	31.536.000	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

- 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  $A[1 \dots n]$ ,  $n \geq 1$

```
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 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  $A[1 \dots n]$ ,  $n \geq 1$
- 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;
```

- 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

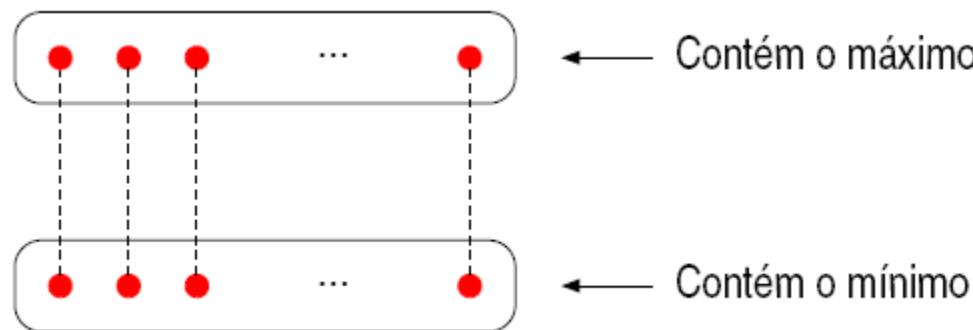
```
procedure MaxMin2 (var A: Vetor; var Max, Min: integer);
var i: integer;
begin
  Max := A[1];  Min := A[1];
  for i := 2 to n do
    if A[i] > Max
      then Max := A[i]
    else if A[i] < Min then Min := A[i];
end;
```

# 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 1<sup>a</sup> posição)
  - Caso médio:  $T(n) = \frac{3n}{2} - \frac{3}{2}$
- Caso médio:
  - A [i] é maior do que 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:
  - 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
  - O máximo é obtido do subconjunto que contém os maiores elementos, a um custo de  $\lceil n/2 \rceil - 1$  comparações
  - 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;
```

# 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)

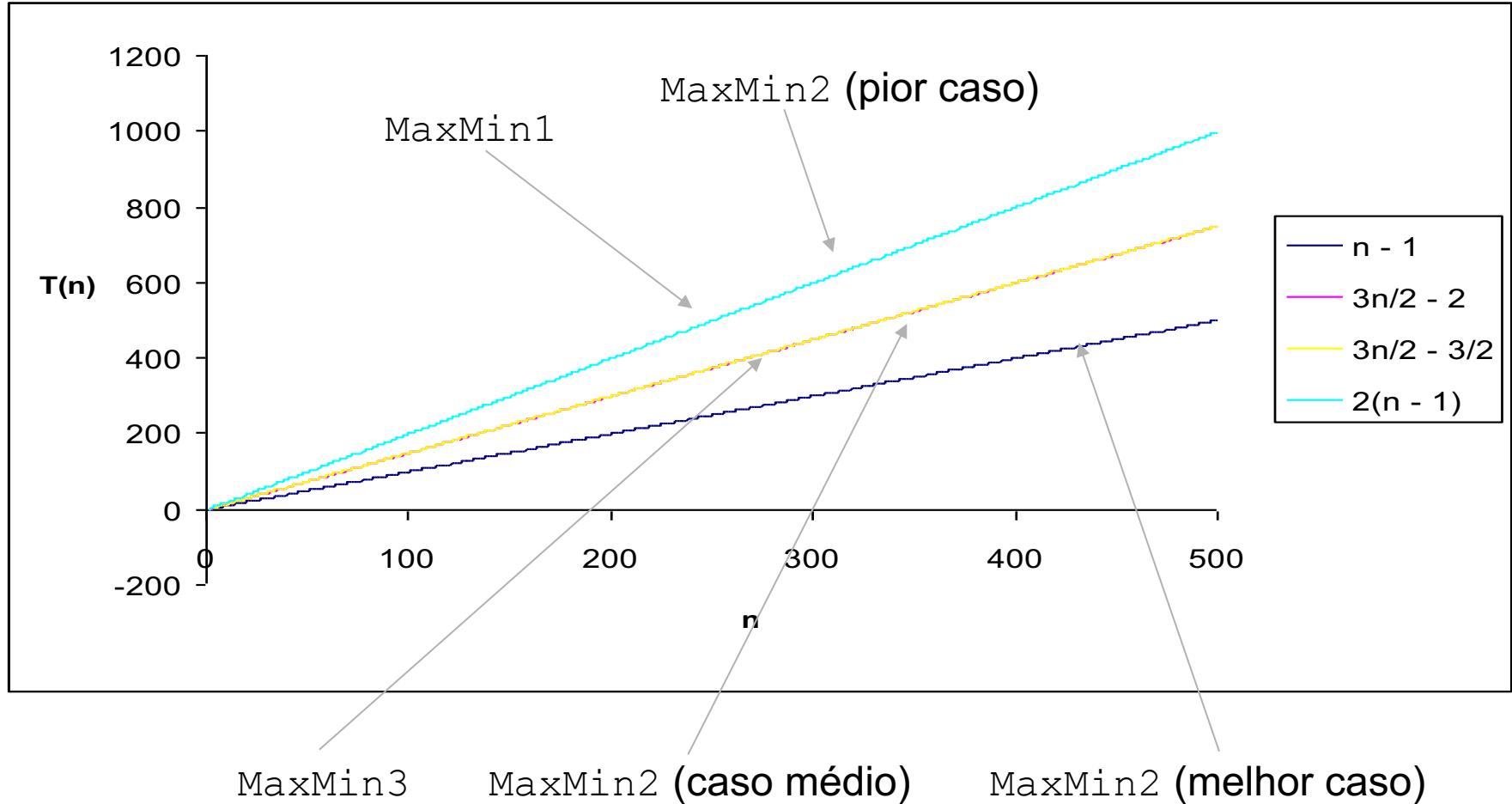
- Os elementos de  $A$  são comparados dois a dois e os elementos maiores são comparados com Max e os elementos menores são comparados com Min
- Quando  $n$  é ímpar, o elemento que está na posição  $A[n]$  é duplicado na posição  $A[n+1]$  para evitar um tratamento de exceção
- Para esta implementação,  $T(n) = \frac{n}{2} + \frac{n-2}{2} + \frac{n-2}{2} = \frac{3n}{2} - 2$  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 ← 1 to n do
    for j ← n downto i+1 do
        if A[j] < A[j-1] then
            A[j] ↔ A[j-1]
```

d)

```
SELECTION-SORT (A)
for i ← 1 to n-1 do
    Min ← i
    for j ← i+1 to n do
        if A[j] < A[Min] then
            Min ← j
    A[Min] ↔ A[i]
```

# **Comportamento Assintótico**

Ziviani – págs. 11 até 19

Cormen – págs. 32 até 49

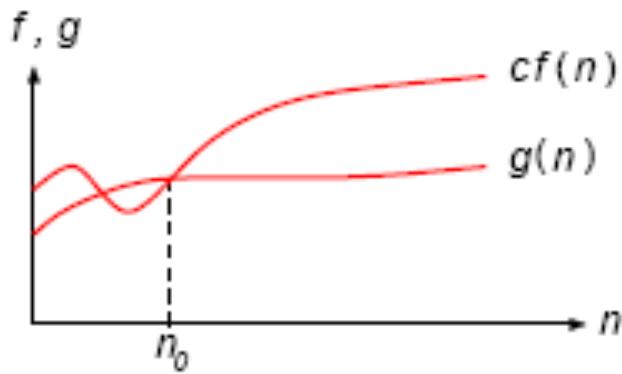
# Comportamento assintótico de funções

- 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

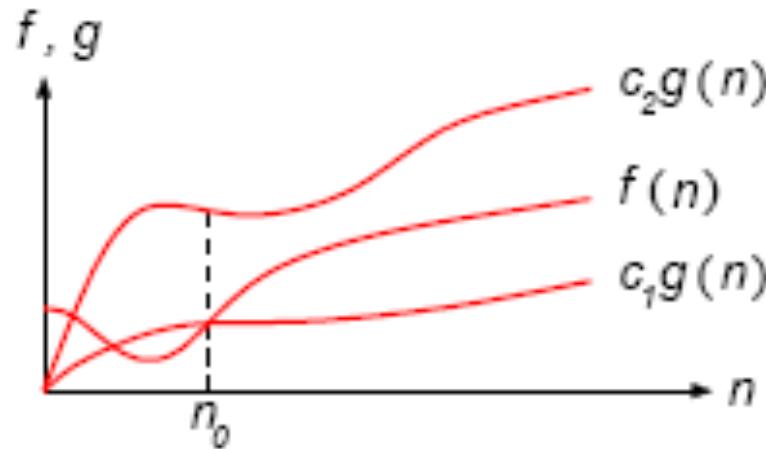
# Custo assintótico de funções

- 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 $\Theta$



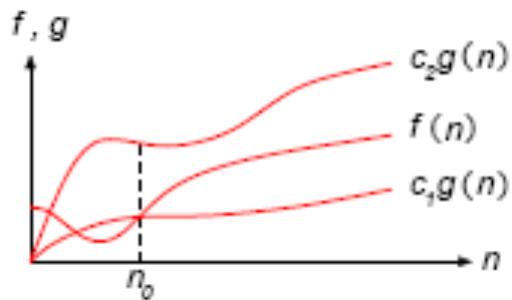
$$f(n) = \Theta(g(n))$$

# 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  $f(n)$

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

$$0 \leq c_1g(n) \leq f(n) \leq c_2g(n), \forall n \geq n_0$$



# Notação Θ: 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:

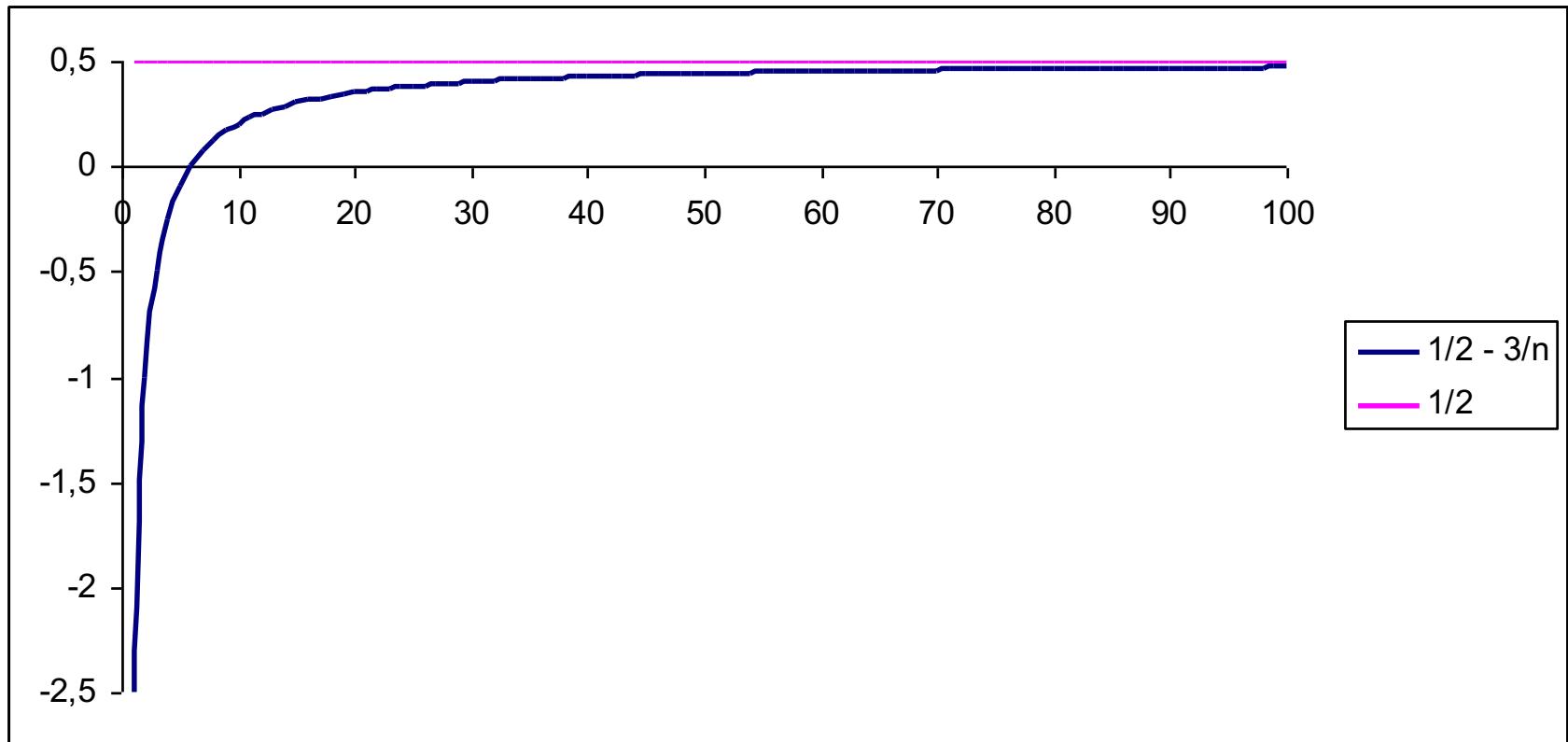
$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

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 Θ: 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

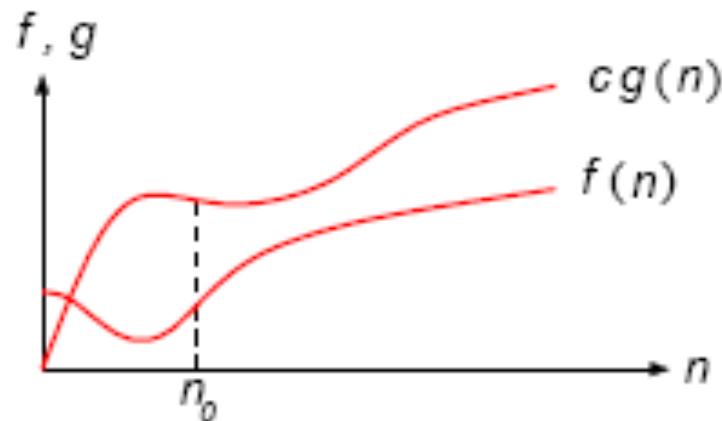
# Exercício

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

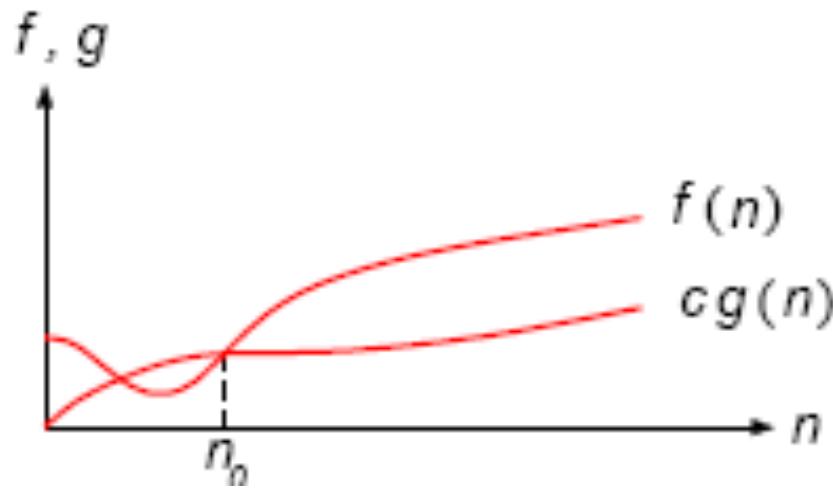
6. Mostre se  $f(n) = O(g(n))$  para os seguintes casos.

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  $\Theta$  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	0,035 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

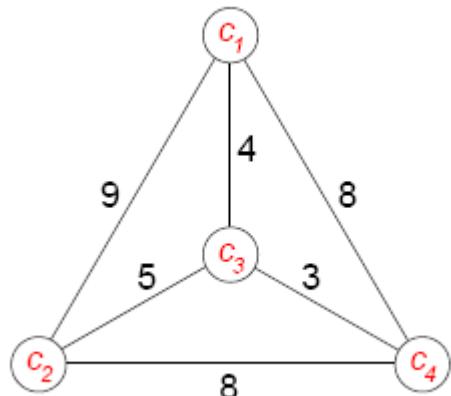
# Algoritmo exponencial × Algoritmo polinomial

- 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

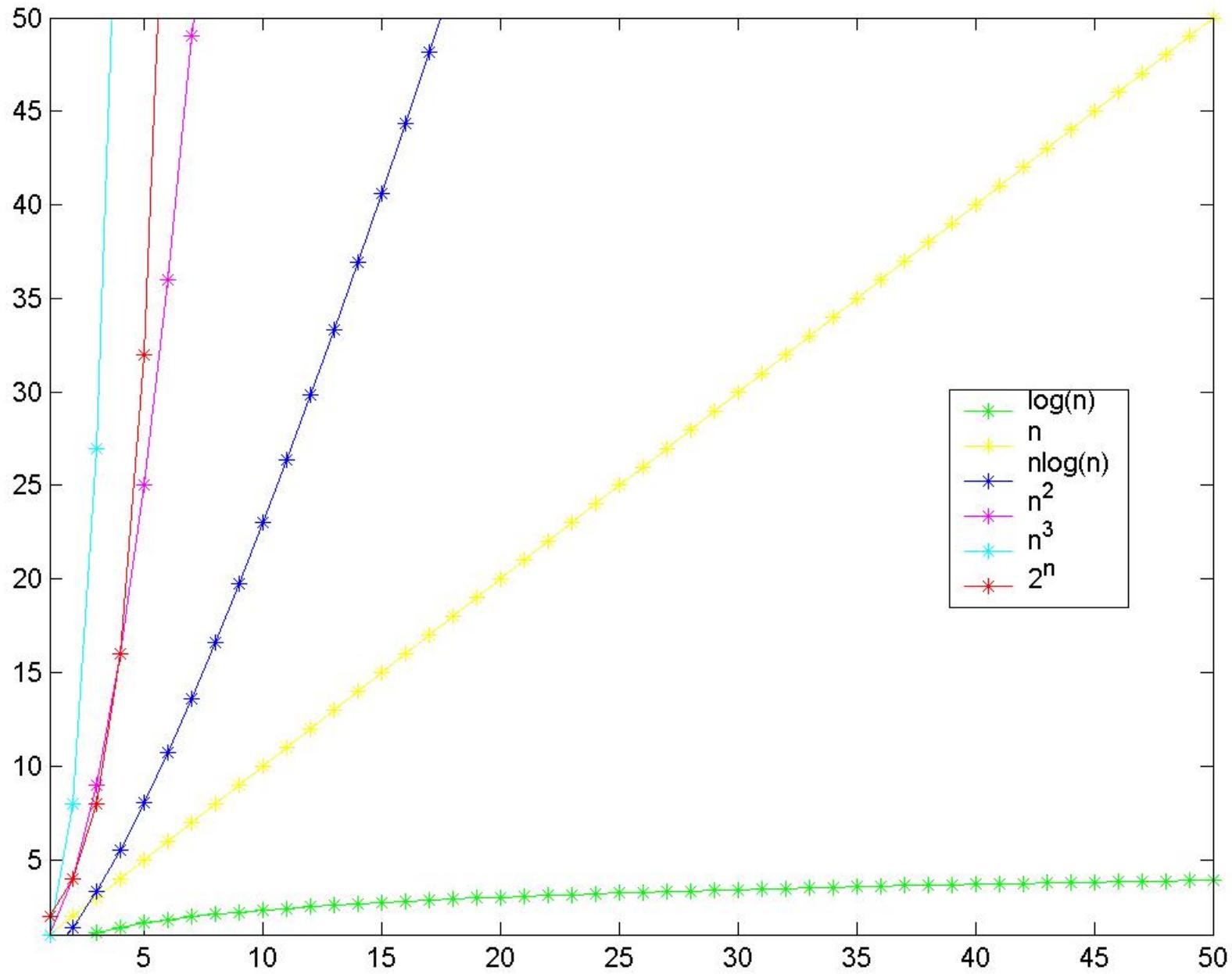
# Hierarquias de funções

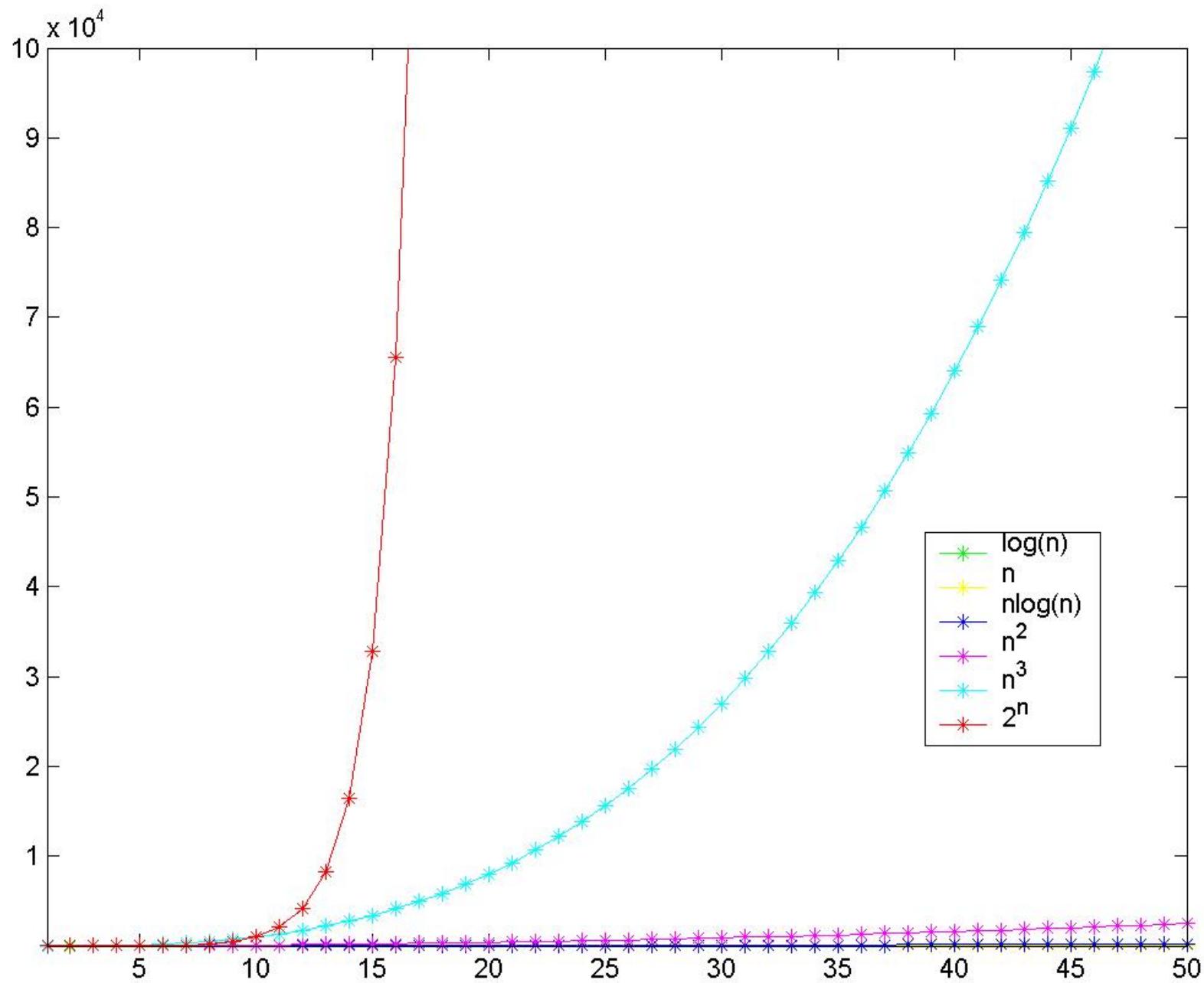
- A seguinte hierarquia de funções pode ser definida do ponto de vista assintótico:

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

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  $\mathbb{A}$ , 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`?

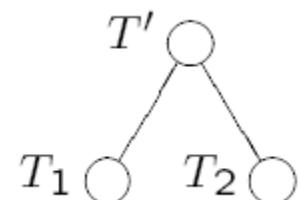
```
void Exercicio3(int n){  
    int i, j, a;  
  
    for (i=0; i<n; i++){  
        if (x[i] > 10)  
            for (j=i+1; j<n; j++)  
                x[j] = x[j] + 2;  
        else {  
            x[i] = 1;  
            j = n-1;  
            while (j >= 0) {  
                x[j] = x[j] - 2;  
                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 Fibonnaci

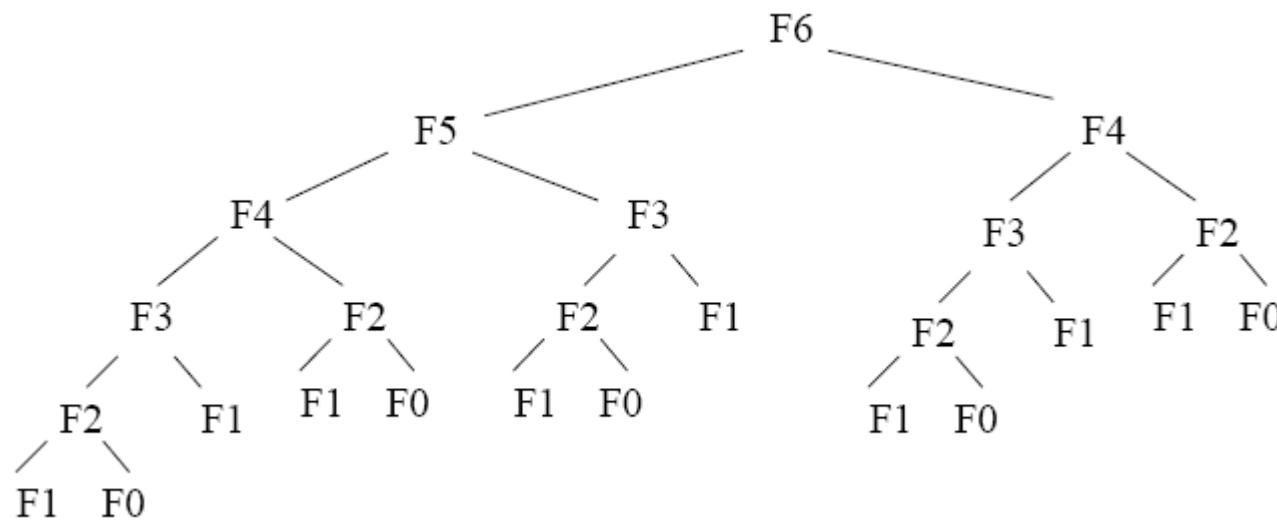
- Série de Fibonnaci:

- $F_n = F_{n-1} + F_{n-2}$        $n > 2$ ,
- $F_0 = F_1 = 1$
- 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  $\phi = (1 + \sqrt{5})/2 = 1,61803\dots$
      - ▶ 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;  
}
```

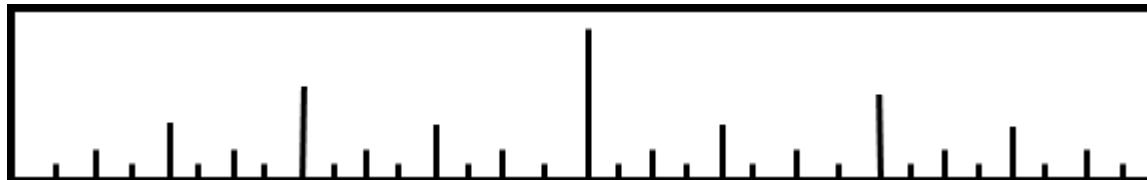
- Complexidade:  $O(n)$
- Conclusão: não usar recursividade cegamente!

# Quando vale a pena usar recursividade?

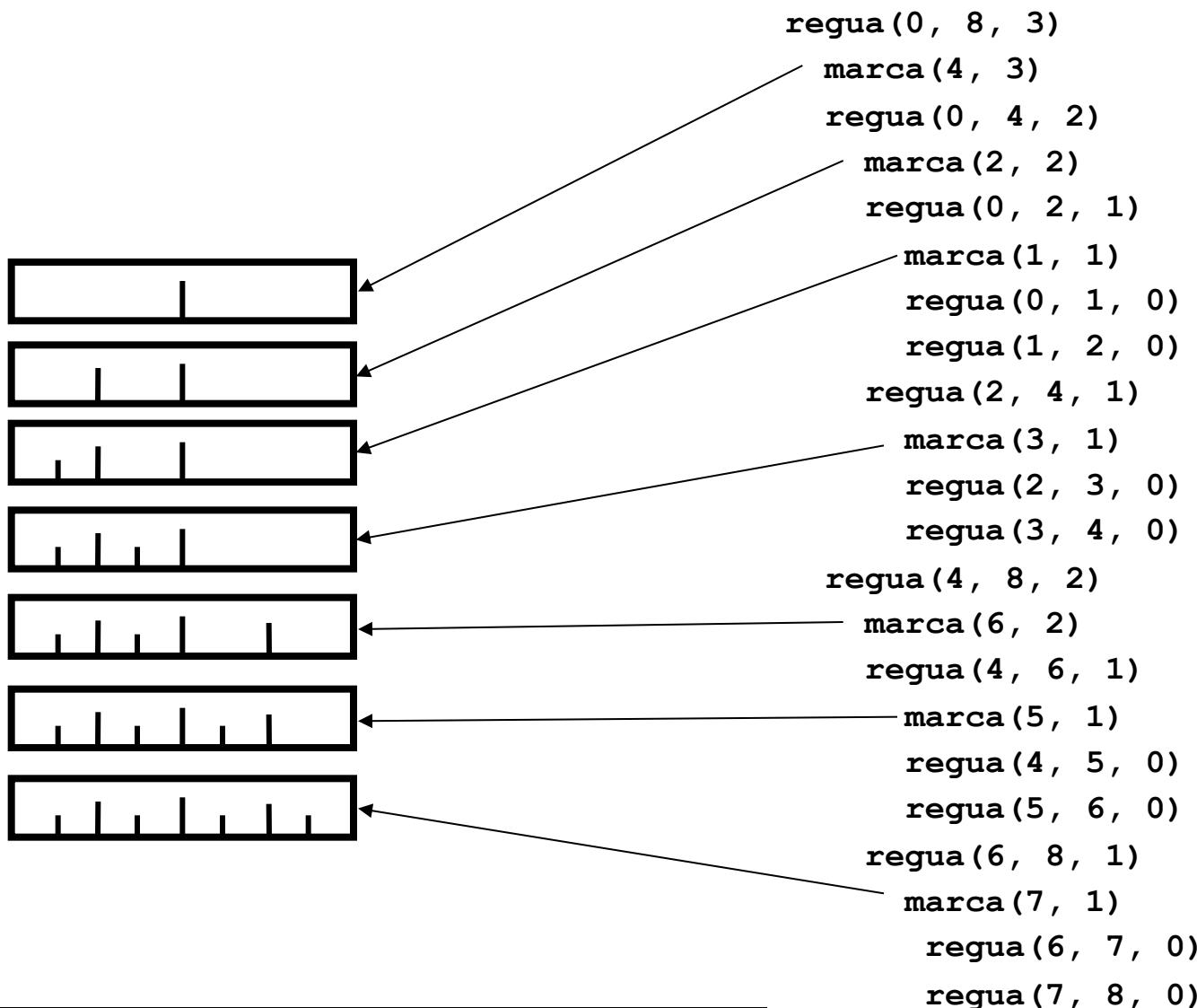
- Recursividade vale a pena para algoritmos complexos, cuja a implementação iterativa é complexa e normalmente requer o uso explícito de uma pilha
  - Dividir para Conquistar (Ex. Quicksort)
  - Caminhamento em Árvores (pesquisa, backtracking)
- Evitar o uso de recursividade quando existe uma solução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

# Exemplo: régua

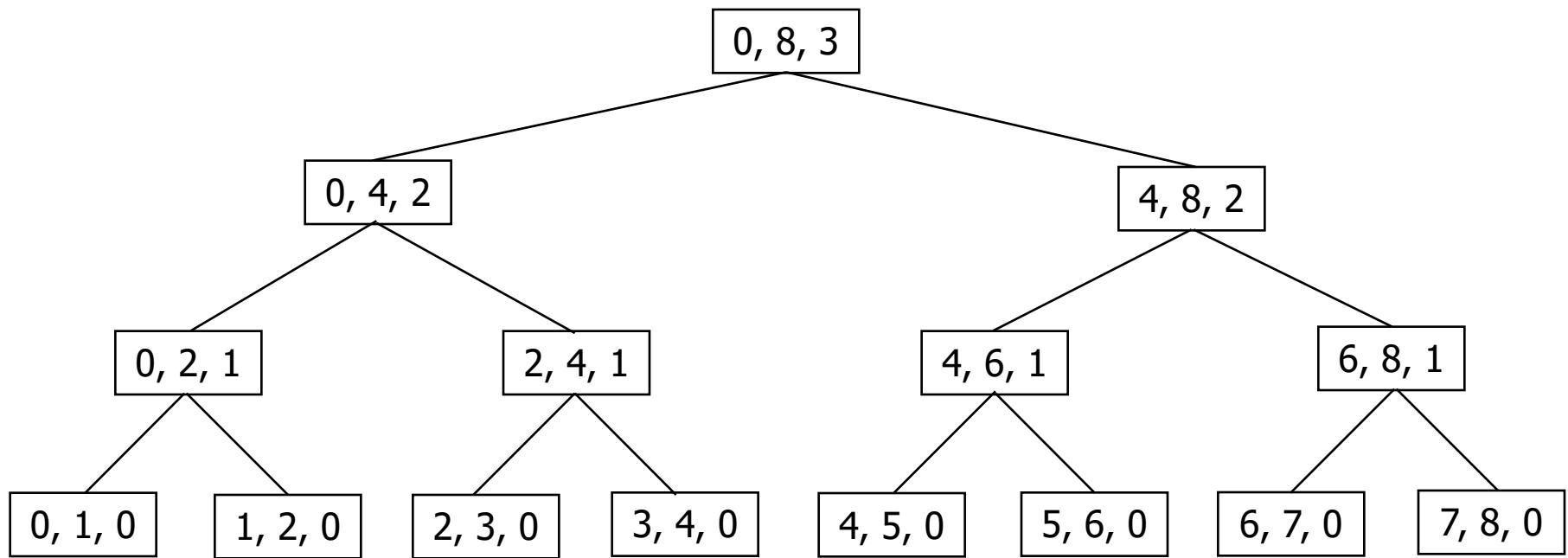
```
void regua(int l, r, h) {  
    int m;  
  
    if (h > 0) {  
        m = (l + r) / 2;  
        marca(m, h);  
        regua(l, m, h - 1);  
        regua(m, r, h - 1);  
    }  
}
```



# 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 ≤ 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

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

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

---

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

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

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

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

---

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

Logo, o programa do exemplo é  $O(n)$

# Exercícios

13. Resolva as seguintes equações de recorrência:

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

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

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

# Teorema Mestre

- Recorrências das 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:

- Se  $f(n) = O(n^{\log_b a - \varepsilon})$  para alguma constante  $\varepsilon > 0$ , então  $T(n) = \Theta(n^{\log_b a})$
- Se  $f(n) = \Theta(n^{\log_b a})$ , então  $T(n) = \Theta(n^{\log_b a} \log n)$
- 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:

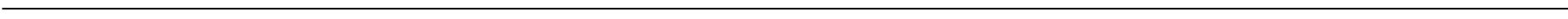
$$T(n) = 4T(n/2) + n$$

**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 2**

**Raquel Mini**  
**[raquelmini@pucminas.br](mailto:raquelmini@pucminas.br)**



# **Força Bruta (Busca Exaustiva ou Enumeração Total)**

# Força Bruta

- Consiste em enumerar todos os possíveis candidatos de uma solução e verificar se cada um satisfaz o problema
- Geralmente possui uma implementação simples e sempre encontrara uma solução se ela existir
- O seu custo computacional é proporcional ao número de candidatos a solução que, em problemas reais, tende a crescer exponencialmente
- É tipicamente usada em problemas cujo tamanho é limitado ou quando não se conhece um algoritmo mais eficiente
- Também pode ser usado quando a simplicidade da implementação é mais importante que a velocidade de execução, como nos casos de aplicações críticas em que os erros de algoritmo possuem sérias consequências

# Força Bruta – Clique

- Considere um conjunto  $P$  de  $n$  pessoas e uma matriz  $M$  de tamanho  $n \times n$ , tal que  $M[i][j] = M[j][i] = 1$ , se as pessoas  $i$  e  $j$  se conhecem e  $M[i][j] = M[j][i] = 0$ , caso contrário
- Problema: existe um subconjunto  $C$  (Clique), de  $r$  pessoas escolhidas de  $P$ , tal que qualquer par de pessoas de  $C$  se conhecem?
- Solução usando força bruta: verificar, para todas as combinações simples (sem repetições)  $C$  de  $r$  pessoas escolhidas entre as  $n$  pessoas do conjunto  $P$ , se todos os pares de pessoas de  $C$  se conhecem

# Força Bruta – Clique

- Considere um conjunto  $P$  de 8 pessoas representado pela matriz abaixo (de tamanho  $8 \times 8$ ):

x	1	2	3	4	5	6	7	8
1	1	0	1	1	1	1	1	0
2	0	1	0	0	1	0	0	1
3	1	0	1	1	0	1	1	1
4	1	0	1	1	1	1	1	1
5	1	1	0	1	1	0	0	0
6	1	0	1	1	0	1	1	1
7	1	0	1	1	0	1	1	0
8	0	1	1	1	0	1	0	1

- Existe um conjunto  $C$  de 5 pessoas escolhidas de  $P$  tal que qualquer par de pessoas de  $C$  se conhecem?

# Força Bruta – Clique

- Existem 56 combinações simples de 5 elementos escolhidos dentre um conjunto de 8 elementos:

1 2 3 4 5	1 2 4 6 8	1 3 5 7 8	2 3 5 6 8
1 2 3 4 6	1 2 4 7 8	1 3 6 7 8	2 3 5 7 8
1 2 3 4 7	1 2 5 6 7	1 4 5 6 7	2 3 6 7 8
1 2 3 4 8	1 2 5 6 8	1 4 5 6 8	2 4 5 6 7
1 2 3 5 6	1 2 5 7 8	1 4 5 7 8	2 4 5 6 8
1 2 3 5 7	1 2 6 7 8	1 4 6 7 8	2 4 5 7 8
1 2 3 5 8	1 3 4 5 6	1 5 6 7 8	2 4 6 7 8
1 2 3 6 7	1 3 4 5 7	2 3 4 5 6	2 5 6 7 8
1 2 3 6 8	1 3 4 5 8	2 3 4 5 7	3 4 5 6 7
1 2 3 7 8	1 3 4 6 7	2 3 4 5 8	3 4 5 6 8
1 2 4 5 6	1 3 4 6 8	2 3 4 6 7	3 4 5 7 8
1 2 4 5 7	1 3 4 7 8	2 3 4 6 8	3 4 6 7 8
1 2 4 5 8	1 3 5 6 7	2 3 4 7 8	3 5 6 7 8
1 2 4 6 7	1 3 5 6 8	2 3 5 6 7	4 5 6 7 8

# Força Bruta – Clique

- Todos os pares de pessoas do subconjunto  $C = \{1, 3, 4, 6, 7\}$  se conhecem:

x	1	3	4	6	7
1	1	1	1	1	1
3	1	1	1	1	1
4	1	1	1	1	1
6	1	1	1	1	1
7	1	1	1	1	1

- Como enumerar todas as combinações simples de  $r$  elementos de um conjunto de tamanho  $n$ ?

# Força Bruta – Combinação

```
#include<iostream>
using namespace std;

void combinacao(int n, int r, int x[], int next, int k){
    int i;
    if (k == r) {
        for (i = 0; i < r; i++)
            cout<<x[i]+1<<" ";
        cout<<endl;
    } else {
        for (i = next; i < n; i++) {
            x[k] = i;
            combinacao(n,r,x,i+1,k+1);
        }
    }
}

int main () {
    int n, r, x[100];
    cout<<"Entre com o valor de n: ";
    cin>>n;
    cout<<"Entre com o valor de r: ";
    cin>>r;
    combinacao(n,r,x,0,0);
    return 0;
}
```

# Força Bruta – Ciclo Hamiltoniano

- Considere um conjunto de  $n$  cidades e uma matriz  $M$  de tamanho  $n \times n$  tal que  $M[i][j] = 1$ , se existir um caminho direto entre as cidades  $i$  e  $j$ , e  $M[i][j] = 0$ , caso contrário
- Problema: existe uma forma de, saindo de uma cidade qualquer, visitar todas as demais cidades, sem passar duas vezes por nenhuma cidade e, no final, retornar para a cidade inicial?
- Se existe uma forma de sair de uma cidade  $x$  qualquer, visitar todas as demais cidades (sem repetir nenhuma) e depois retornar para  $x$ , então existe um ciclo Hamiltoniano e qualquer cidade do ciclo pode ser usada como ponto de partida

# Força Bruta – Ciclo Hamiltoniano

- Como vimos, qualquer cidade pode ser escolhida como cidade inicial. Sendo assim, vamos escolher, arbitrariamente a cidade  $n$  como ponto de partida
- Solução usando força bruta: testar todas as permutações das  $n-1$  primeiras cidades, verificando se existe um caminho direto entre a cidade  $n$  e a primeira da permutação, assim como um caminho entre todas as cidades consecutivas da permutação e, por fim, um caminho direto entre a última cidade da permutação e a cidade  $n$
- Ciclo Hamiltoniano:  $n \rightsquigarrow [p_1 \rightsquigarrow p_2 \rightsquigarrow p_3 \rightsquigarrow \dots \rightsquigarrow p_{n-1}] \rightsquigarrow n$

# Força Bruta – Ciclo Hamiltoniano

- Considere um conjunto de 8 cidades representado pela matriz abaixo:

x	1	2	3	4	5	6	7	8
1	0	0	1	0	1	1	1	0
2	0	1	0	0	1	0	0	1
3	1	0	1	1	0	1	1	0
4	0	0	1	1	0	0	1	0
5	1	1	0	1	1	0	0	0
6	0	0	1	1	0	0	1	1
7	1	0	0	1	0	1	1	1
8	0	1	1	1	0	1	0	1

- Existe uma forma de, a partir da cidade 8, visitar todas as demais cidades, sem repetir nenhuma e, ao final, retornar para a cidade 8?

# Força Bruta – Ciclo Hamiltoniano

- Existem 5040 permutações das 7 primeiras cidades da lista original:

1 2 3 4 5 6 7	...	7 6 5 2 3 4 1
1 2 3 4 5 7 6	3 6 4 5 1 7 2	7 6 5 2 4 1 3
1 2 3 4 6 5 7	3 6 4 5 2 1 7	7 6 5 2 4 3 1
1 2 3 4 6 7 5	3 6 4 5 2 7 1	7 6 5 3 1 2 4
1 2 3 4 7 5 6	3 6 4 5 7 1 2	7 6 5 3 1 4 2
1 2 3 4 7 6 5	3 6 4 5 7 2 1	7 6 5 3 2 1 4
1 2 3 5 4 6 7	3 6 4 7 1 2 5	7 6 5 3 2 4 1
1 2 3 5 4 7 6	3 6 4 7 1 5 2	7 6 5 3 4 1 2
1 2 3 5 6 4 7	3 6 4 7 2 1 5	7 6 5 3 4 2 1
1 2 3 5 6 7 4	3 6 4 7 2 5 1	7 6 5 4 1 2 3
1 2 3 5 7 4 6	3 6 4 7 5 1 2	7 6 5 4 1 3 2
1 2 3 5 7 6 4	3 6 4 7 5 2 1	7 6 5 4 2 1 3
1 2 3 6 4 5 7	3 6 5 1 2 4 7	7 6 5 4 2 3 1
1 2 3 6 4 7 5	3 6 5 1 2 7 4	7 6 5 4 3 1 2
1 2 3 6 5 4 7	...	7 6 5 4 3 2 1

- Como enumerar todas as permutações de  $n$  valores distintos?

# Força Bruta – Permutação

```
#include<iostream>
using namespace std;

void permutacao(int n, int x[], bool used[], int k) {
    int i;
    if (k == n) {
        for (i = 0; i < n; i++)
            cout<<x[i]+1<<" ";
        cout<<endl;
    } else {
        for (i = 0; i < n; i++) {
            if (!used[i]) {
                used[i] = true;
                x[k] = i;
                permutacao(n, x, used, k+1);
                used[i] = false;
            }
        }
    }
}

int main () {
    int i, n, x[100];
    bool used[100];
    cout<<"Entre com o valor de n: ";
    cin>>n;
    for (i = 0; i < n; i++)
        used[i] = false;
    permutacao(n, x, used, 0);
    return 0;
}
```

# Incremental

# Incremental

- A ordenação por inserção utiliza uma abordagem incremental: tendo ordenado o subarranjo  $A[1..j-1]$ , inserimos o elemento isolado  $A[j]$  em seu lugar apropriado, formando o subarranjo ordenado  $A[1..j]$ .



# Incremental

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

# Exercícios

1. Implemente um algoritmo que enumere todos os arranjos de tamanho  $r$  dentre um conjunto de  $n$  elementos.

# Algoritmos Tentativa e Erros *(Backtracking)*

Ziviani – págs. 44 até 48

# *Backtracking*

- Algoritmo para encontrar todas (ou algumas) soluções de um problema computacional, que incrementalmente constrói candidatas de soluções e abandona uma candidata parcialmente construída tão logo quanto for possível determinar que ela não pode gerar uma solução válida
- Pode ser aplicado para problemas que admitem o conceito de “solução candidata parcial” e que exista um teste relativamente rápido para verificar se uma candidata parcial pode ser completada como uma solução válida

# *Backtracking*

- Quando aplicável, *backtracking* é frequentemente muito mais rápido que algoritmos de força bruta, já que ele pode eliminar um grande número de soluções inválidas com um único teste
- Enquanto algoritmos de força bruta geram todas as possíveis soluções e só depois verificam se elas são válidas, *backtracking* só gera soluções válidas

# Backtracking – Passeio do Cavalo

- Tabuleiro com  $n \times n$  posições: cavalo se movimenta segundo regras do xadrez
- Problema: partindo da posição  $(x_0, y_0)$ , encontrar, se existir, um passeio do cavalo que visita todos os pontos do tabuleiro uma única vez

Tenta um próximo movimento

TENTA

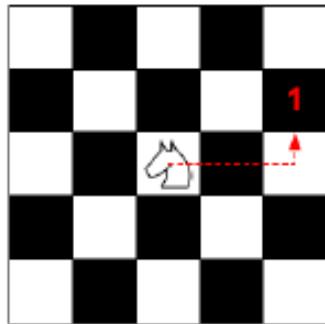
- 1 Inicializa seleção de movimentos
- 2 **repeat**
- 3     Seleciona próximo candidato ao movimento
- 4     **if** aceitável
- 5         **then** Registra movimento
- 6             **if** tabuleiro não está cheio
- 7                 **then** Tenta novo movimento
- 8                     **if** não é bem sucedido
- 9                         **then** Apaga registro anterior
- 10   **until** (movimento bem sucedido)  $\vee$  (acabaram-se candidatos ao movimento)

# *Backtracking – Passeio do Cavalo*

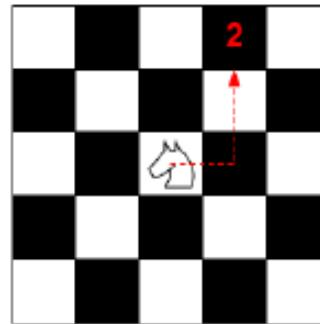
- O tabuleiro pode ser representado por uma matriz  $n \times n$
- A situação de cada posição pode ser representada por um inteiro para recordar o histórico das ocupações:
  - $t[x][y] = 0$ , campo  $\langle x, y \rangle$  não visitado
  - $t[x][y] = i$ , campo  $\langle x, y \rangle$  visitado no  $i$ -ésimo movimento,  $1 \leq i \leq n^2$

# Backtracking – Passeio do Cavalo

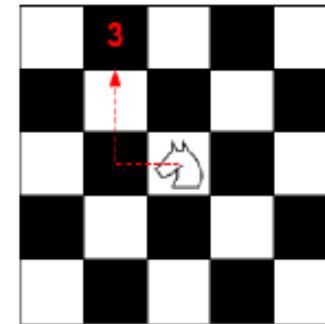
- Regras do xadrez para os movimentos do cavalo:



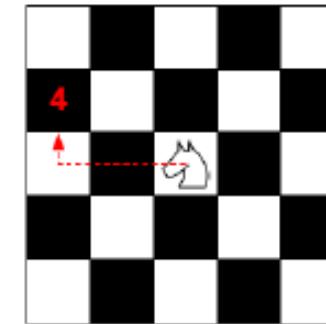
2Dir e 1Cima



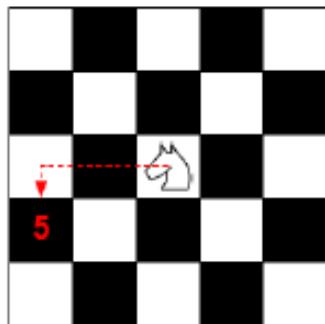
1Dir e 2Cima



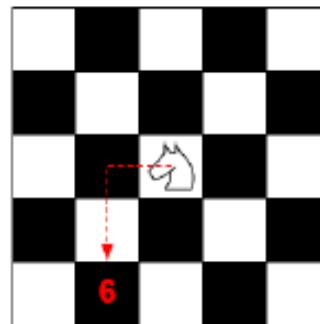
1Esq e 2Cima



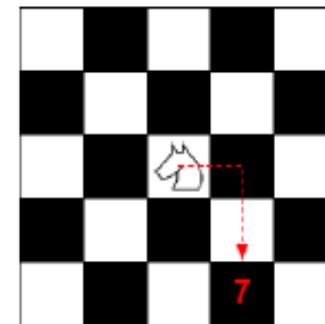
2Esq e 1Cima



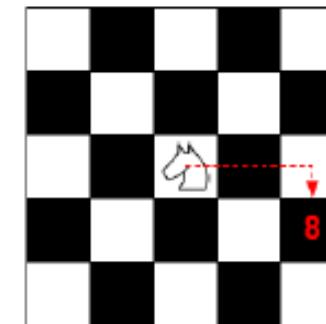
2Esq e 1Baixo



1Esq e 2Baixo



1Dir e 2Baixo



2Dir e 1Baixo

# Backtracking – Passeio do Cavalo

PASSEIO DO CAVALO( $n$ )

- ▷ Parâmetro:  $n$  (tamanho do lado do tabuleiro)
- ▷ Variáveis auxiliares:

$i, j$   
 $t[1..n, 1..n]$

$q$   
 $s$

$h[1..8], v[1..8]$

```
1  $s \leftarrow \{1, 2, 3, 4, 5, 6, 7, 8\}$ 
2  $h[1..8] \leftarrow [2, 1, -1, -2, -2, -1, 1, 2]$ 
3  $v[1..8] \leftarrow [1, 2, 2, 1, -1, -2, -2, -1]$ 
4 for  $i \leftarrow 1$  to  $n$ 
    do for  $j \leftarrow 1$  to  $n$ 
        do  $t[i, j] \leftarrow 0$ 
    end for
     $t[1, 1] \leftarrow 1$ 
    TENTA( $2, 1, 1, q$ )
    if  $q$ 
        then print Solução
    else print Não há solução
```

- ▷ Contadores
- ▷ Tabuleiro de  $n \times n$
- ▷ Indica se achou uma solução
- ▷ Movimentos identificados por um  $n^{\circ}$
- ▷ Existem oito movimentos possíveis
- ▷ Conjunto de movimentos
- ▷ Movimentos na horizontal
- ▷ Movimentos na vertical
- ▷ Inicializa tabuleiro
- ▷ Escolhe uma casa inicial do tabuleiro
- ▷ Tenta o passeio usando *backtracking*
- ▷ Achou uma solução?

# Backtracking – Passeio do Cavalo

TENTA( $i, x, y, q$ )

▷ Parâmetros:  $i$  ( $i$ -ésima casa);  $x, y$  (posição no tabuleiro);  $q$  (achou solução?)

▷ Variáveis auxiliares:  $xn, yn, m, q1$

```
1   $m \leftarrow 0$ 
2  repeat
3       $m \leftarrow m + 1$ 
4       $q1 \leftarrow \text{false}$ 
5       $xn \leftarrow x + h[m]$ 
6       $yn \leftarrow y + v[m]$ 
7      if ( $xn \in s$ )  $\wedge$  ( $yn \in s$ )
8          then if  $t[xn, yn] = 0$ 
9              then  $t[xn, yn] \leftarrow i$ 
10             if  $i < n^2$ 
11                 then TENTA( $i + 1, xn, yn, q1$ )
12                 if  $\neg q1$ 
13                     then  $t[xn, yn] \leftarrow 0$ 
14                 else  $q1 \leftarrow \text{true}$ 
15 until  $q1 \vee (m = 8)$ 
16  $q \leftarrow q1$ 
```

# Backtracking – Passeio do Cavalo

- Resultado do Passeio do Cavalo em um tabuleiro 8 x 8

1	60	39	34	31	18	9	64
38	35	32	61	10	63	30	17
59	2	37	40	33	28	19	8
36	49	42	27	62	11	16	29
43	58	3	50	41	24	7	20
48	51	46	55	26	21	12	15
57	44	53	4	23	14	25	6
52	47	56	45	54	5	22	13

# Backtracking – Labirinto

- Dado um labirinto representado por uma matriz de tamanho  $n \times m$ , uma posição inicial  $p_i = (x_i; y_i)$  e uma posição final  $p_f = (x_f; y_f)$ , tal que  $p_i \neq p_f$ , determinar se existe um caminho entre  $p_i$  e  $p_f$
- Podemos representar o labirinto como uma matriz  $M$  tal que:

$$M[x, y] = \begin{cases} -2, & \text{se a posição } (x, y) \text{ representa uma parede} \\ -1, & \text{se a posição } (x, y) \text{ não pertence ao caminho} \\ i, & \text{tal que } i \geq 0, \text{ se a posição } (x, y) \text{ pertence ao caminho} \end{cases}$$

- Neste caso, vamos supor que o labirinto é cercado por paredes, eventualmente apenas com exceção do local designado como saída

# Backtracking – Labirinto

- A figura abaixo mostra um labirinto de tamanho 8 x 8

X	X	X	X	X	X	X	X
X	•						X
X	X		X				X
X			X	X	X		X
X		X	X				X
X		X				X	X
X				X			X
X	X	X	X	X	X	o	X

X: parede/obstáculo

•: posição inicial

o: posição final (saída do labirinto)

# Backtracking – Labirinto

- Caminho encontrado usando a seguinte ordem de busca:
  - para esquerda
  - para baixo
  - para direita
  - para cima

X	X	X	X	X	X	X	X
X	00	01					X
X	X	02	X				X
X	04	03	X	X	X		X
X	05	X	X				X
X	06	X	10	11	12	X	X
X	07	08	09	X	13	14	X
X	X	X	X	X	X	15	X

# Backtracking – Labirinto

- Caminho encontrado usando a seguinte ordem de busca:
  - para direita
  - para baixo
  - para esquerda
  - para cima

X	X	X	X	X	X	X	X
X	00	01	02	03	04	05	X
X	X		X			06	X
X			X	X	X	07	X
X		X	X		09	08	X
X		X			10	X	X
X				X	11	12	X
X	X	X	X	X	X	13	X

# Backtracking – Labirinto

```
#include<iostream>
#include <iomanip>
#define MAX 10
using namespace std;
void imprimeLabirinto(int M[MAX][MAX], int n, int m) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            if (M[i][j] == -2) cout<<" XX";
            if (M[i][j] == -1) cout<<"   ";
            if (M[i][j] >= 0) cout<<" " <<setw(2)<<M[i][j];
        }
        cout<<"\n";
    }
}
void obtemLabirinto(int M[MAX][MAX], int &n, int &m, int &Li, int &Ci, int &Lf, int &Cf) {
    int i, j, d;
    cin>>n; cin>>m; /* dimensoes do labirinto */
    cin>>Li; cin>>Ci; /* coordenadas da posicao inicial */
    cin>>Lf; cin>>Cf; /* coordenadas da posicao final (saida) */
    /* labirinto: 1 = parede ou obstaculo 0 = posicao livre */
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++) {
            cin>>d;
            if (d == 1)
                M[i][j] = -2;
            else
                M[i][j] = -1;
        }
}
```

# Backtracking – Labirinto

```
int labirinto(int M[MAX][MAX], int deltaL[], int deltaC[], int Li, int Ci, int Lf, int Cf) {
    int L, C, k, passos;
    if ((Li == Lf) && (Ci == Cf)) return M[Li][Ci];
    /* testa todos os movimentos a partir da posicao atual */
    for (k = 0; k < 4; k++) {
        L = Li + deltaL[k];
        C = Ci + deltaC[k];
        /* verifica se o movimento eh valido e gera uma solucao factivel */
        if (M[L][C] == -1) {
            M[L][C] = M[Li][Ci] + 1;
            passos = labirinto(M, deltaL, deltaC, L, C, Lf, Cf);
            if (passos > 0) return passos;
        }
    }
    return 0;
}
int main() {
    int M[MAX][MAX], resposta, n, m, Li, Ci, Lf, Cf;
    // define os movimentos validos no labirinto
    int deltaL[4] = { 0, +1, 0, -1};
    int deltaC[4] = {+1, 0, -1, 0};
    // obtém as informações do labirinto
    obtemLabirinto(M, n, m, Li, Ci, Lf, Cf);
    M[Li - 1][Ci - 1] = 0; /* define a posição inicial no tabuleiro */
    /* tenta encontrar um caminho no labirinto */
    resposta = labirinto(M, deltaL, deltaC, Li - 1, Ci - 1, Lf - 1, Cf - 1);
    if (resposta == 0)
        cout<<"Nao existe solucao.\n";
    else {
        cout<<"Existe uma solucao em "<<resposta<<" passos.\n";
        imprimeLabirinto(M, n, m);
    }
    return 0;
}
```

# *Backtracking – Labirinto*

- Exemplo de entrada:

8	8						
2	2						
8	7						
1	1	1	1	1	1	1	1
1	0	1	0	0	0	0	1
1	0	0	0	1	0	0	1
1	0	0	1	1	0	0	1
1	0	1	1	0	0	0	1
1	0	0	1	1	1	1	1
1	0	0	0	0	0	0	1
1	1	1	1	1	1	0	1

# *Backtracking – Labirinto*

- Exemplo de saída:

```
Existe uma solucao em 13 passos .
XX XX XX XX XX XX XX XX
XX 0 XX 4 5 6 7 XX
XX 1 2 3 XX 13 8 XX
XX 4 3 XX XX 12 9 XX
XX 5 XX XX 12 11 10 XX
XX 6 7 XX XX XX XX XX
XX 8 9 10 11 12 XX
XX XX XX XX XX XX 13 XX
```

# *Backtracking* – Problemas de Otimização

- Muitas vezes não estamos apenas interessados em encontrar uma solução qualquer, mas em encontrar uma solução ótima (segundo algum critério de otimalidade pré-estabelecido)
- Por exemplo, no problema do labirinto, ao invés de determinar se existe um caminho entre o ponto inicial e o final (saída), podemos estar interessados em encontrar uma solução que usa o menor número possível de passos

# *Branch and Bound*

# *Branch and Bound*

- Refere-se a um tipo de algoritmo usado para encontrar soluções ótimas para vários problemas de otimização
- Problemas de otimização podem ser tanto de maximização quanto de minimização
- Consiste em uma enumeração sistemática de todos os candidatos à solução, com eliminação de uma candidata parcial quando uma dessas duas situações for detectada (considerando um problema de minimização):
  - A candidata parcial é incapaz de gerar uma solução válida (teste similar ao realizado pelo método *backtracking*)
  - A candidata parcial é incapaz de gerar uma solução ótima, considerando o valor da melhor solução encontrada até então (limitante superior) e o custo ainda necessário para gerar uma solução a partir da solução candidata atual (limitante inferior)

# *Branch and Bound*

- O desempenho de um programa *branch and bound* está fortemente relacionado à qualidade dos seus limitantes inferiores e superiores
  - Quando mais precisos forem esses limitantes, menos soluções parciais serão consideradas e mais rápido o programa encontrará a solução ótima
  - O nome *branch and bound* refere-se às duas fases do algoritmo:
    - ▶ *Branch*: testar todas as ramificações de uma solução candidata parcial
    - ▶ *Bound*: limitar a busca por soluções sempre que detectar que o atual ramo da busca é infrutífero

# *Branch and Bound* – Labirinto

- Podemos alterar o programa visto anteriormente para encontrar um caminho ótimo num labirinto usando a técnica *branch and bound*
- Podemos inicialmente notar que se um caminho parcial já usou tantos passos quanto o melhor caminho completo previamente descoberto, então este caminho parcial pode ser descartado
- Mais do que isso, se o número de passos do caminho parcial mais o número de passos mínimos necessários entre a posição atual e a saída (desconsiderando eventuais obstáculos) for maior ou igual ao número de passos do melhor caminho previamente descoberto, então este caminho parcial também pode ser descartado

# *Branch and Bound – Labirinto*

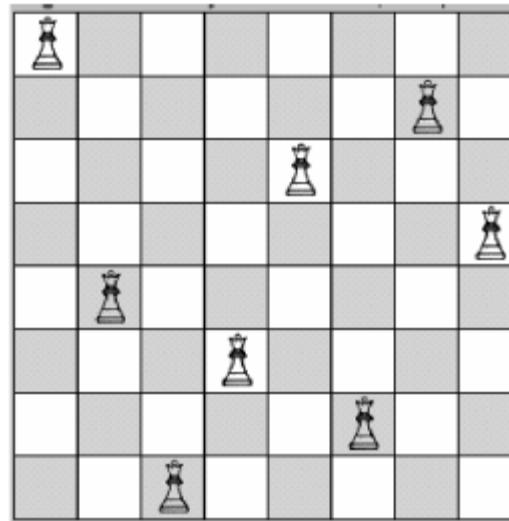
```
void labirinto(int M[MAX][MAX], int deltaL[], int deltaC[], int Li, int Ci, int Lf, int Cf, int &min) {
    int L, C, k;
    if ((Li == Lf) && (Ci == Cf)) {
        if (M[Lf][Cf] < min)
            min = M[Li][Ci];
    } else {
        /* testa todos os movimentos a partir da posicao atual */
        for (k = 0; k < 4; k++) {
            L = Li + deltaL[k];
            C = Ci + deltaC[k];
            /* verifica se o movimento eh valido e pode gerar uma solucao otima */
            if ((M[L][C] == -1) || (M[L][C] > M[Li][Ci] + 1)) {
                M[L][C] = M[Li][Ci] + 1;
                if (M[L][C] < min)
                    labirinto(M, deltaL, deltaC, L, C, Lf, Cf, min);
            }
        }
    }
}
```

# *Branch and Bound – Labirinto*

```
int main() {
    int M[MAX][MAX], n, m, Li, Ci, Lf, Cf,min;
    // define os movimentos validos no labirinto
    int deltaL[4] = { 0, +1, 0, -1};
    int deltaC[4] = {+1, 0, -1, 0};
    // obtém as informações do labirinto
    obtemLabirinto(M,n,m,Li,Ci,Lf,Cf);
    M[Li - 1][Ci - 1] = 0; /* define a posição inicial no tabuleiro */
    /* tenta encontrar um caminho no labirinto */
    min = INT_MAX;
    labirinto(M, deltaL, deltaC, Li - 1, Ci - 1, Lf - 1, Cf - 1,min);
    if (min == 0)
        cout<<"Nao existe solucao.\n";
    else {
        cout<<"Existe uma solucao em "<<min<<" passos.\n";
        imprimeLabirinto(M, n, m);
    }
    return 0;
}
```

# Exercício

2. Proponha um algoritmo para a solução do problema das 8 rainhas utilizando *backtracking*. Dado um tabuleiro de xadrez (com 8 x 8 casas), o objetivo é distribuir 8 rainhas sobre este tabuleiro de modo que nenhuma delas fique em posição de ser atacada por outra rainha.



# **Divisão e Conquista**

Ziviani – págs. 48 até 51

Cormen – págs. 21 até 28

# Divisão e Conquista

- O paradigma divisão e conquista consiste em dividir o problema a ser resolvido em partes menores, encontrar soluções para as partes, e então combinar as soluções obtidas em uma solução global
- O paradigma divisão e conquista envolve três passos em cada nível de recursão:
  - **Dividir** o problema em um determinado número de subproblemas
  - **Conquistar** os subproblemas, resolvendo-os recursivamente. Se o tamanho dos subproblemas forem pequenos o bastante, basta resolver os subproblemas de maneira direta
  - **Combinar** as soluções dadas aos problemas a fim de formar a solução para o problema original

# Divisão e Conquista - Esquema

```
void divide_and_conquer(Problem P, Solution S) {  
    if (small(P))  
        S = compute_solution(P);  
    else {  
        divide P em problemas menores do mesmo  
        tipo do original, P1, P2, ..., Pk; ← DIVIDIR  
        divide_and_conquer(P1, S1); ← CONQUISTAR  
        .  
        .  
        .  
        divide_and_conquer(Pk, Sk); ← Resolvendo  
        recombine S1, S2, ..., Sk em S, uma solução para P;  
    }  
}
```

↑ COMBINAR

**CONQUISTAR**  
**Resolvendo**  
**subproblemas**  
**recursivamente**

# Divisão e Conquista

- Análise de Complexidade:

- Se o tamanho do problema for pequeno o bastante  $n \leq c$ , para alguma constante  $c$ , a solução direta demorará um tempo constante  $\Theta(1)$
- Vamos supor que o problema seja dividido em  $a$  subproblemas, cada um dos quais com  $1/b$  do tamanho do problema original
- $D(n)$  é o tempo para dividir o problema em subproblemas
- $C(n)$  tempo para combinar as soluções dadas aos subproblemas na solução para o problema original

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{caso contrário} \end{cases}$$

# Exemplo: Maior e Menor Elemento

- Exemplo: encontrar o maior e o menor elemento de um vetor de inteiros,  $A[0..n-1]$ ,  $n \geq 1$
- Cada chamada de `MaxMin4` atribui à `max` e `min` o maior e o menor elemento em  $A[\text{esq}], A[\text{esq}+1], \dots, A[\text{dir}]$ , respectivamente

# Exemplo: Maior e Menor Elemento

```
void MaxMin4(int A[], int esq, int dir, int &min, int &max) {  
    int min1, min2, max1, max2, meio;  
  
    if (dir-esq <= 1) {  
        if (A[esq] < A[dir]) {  
            min = A[esq];  
            max = A[dir];  
        }  
        else {  
            min = A[dir];  
            max = A[esq];  
        }  
    }  
    else {  
        meio = (esq+dir)/2;  
        MaxMin4(A,esq,meio,min1,max1);  
        MaxMin4(A,meio+1,dir,min2,max2);  
        max = (max1 > max2) ? max1 : max2;  
        min = (min1 < min2) ? min1 : min2;  
    }  
}
```

# Análise do Maior e Menor Elemento

- Seja  $T(n)$  o número de comparações entre os elementos de A

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

- $T(n) = \frac{3n}{2} - 2$  para o melhor caso, pior caso e caso médio

# Análise do Maior e Menor Elemento

- Conforme mostrado no início do curso, o algoritmo dado neste exemplo é ótimo
- Entretanto, ele pode ser pior que o MaxMin3 pois, a cada chamada recursiva, salva `esq`, `dir`, `min` e `max` além do endereço de retorno da chamada para o procedimento
- Além disso, uma comparação adicional é necessária a cada chamada recursiva para verificar se  $\text{dir} - \text{esq} \leq 1$
- $n+1$  deve ser menor que a metade do maior inteiro que pode ser representado pelo compilador, para não provocar *overflow* na operação `esq+dir`

# Exemplo: Mergesort

- O algoritmo de ordenação Mergesort obedece ao paradigma de dividir e conquistar
  - **DIVIDIR**: divide a sequência de  $n$  elementos a serem ordenados em duas subsequências de  $n/2$  elementos cada uma
  - **CONQUISTAR**: ordena as duas subsequências recursivamente, utilizando o MERGE-SORT
  - **COMBINAR**: faz a intercalação das duas sequências ordenadas, de modo a produzir a resposta ordenada (função MERGE)

# Exemplo: Mergesort

```
MERGE-SORT (A, p, r)
```

```
if p < r then
```

$$q \leftarrow \lfloor (p + r) / 2 \rfloor$$

```
MERGE-SORT (A, p, q)
```

```
MERGE-SORT (A, q+1, r)
```

```
MERGE (A, p, q, r)
```

DIVIDIR

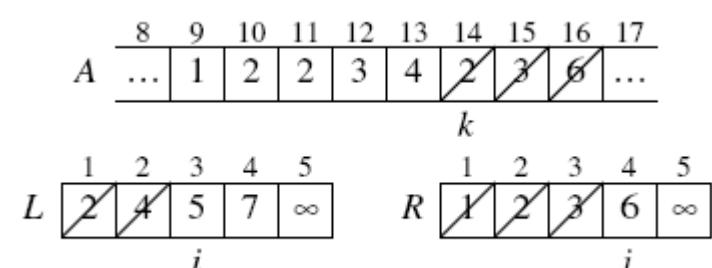
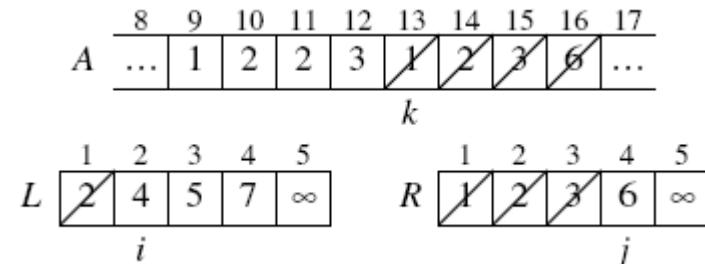
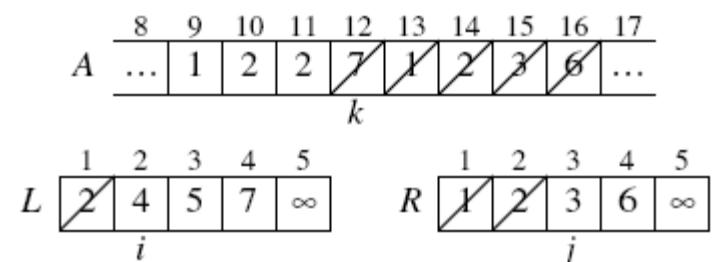
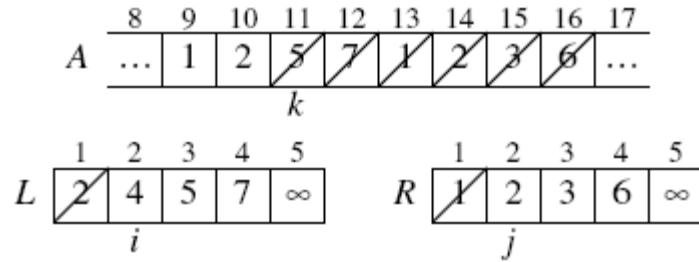
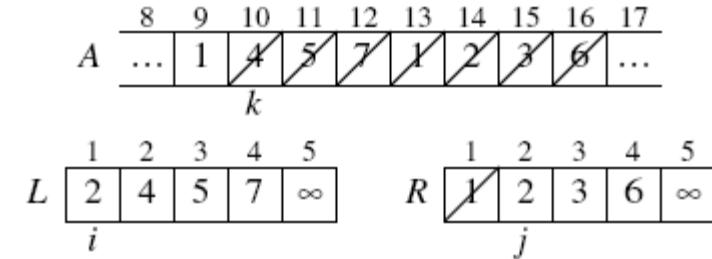
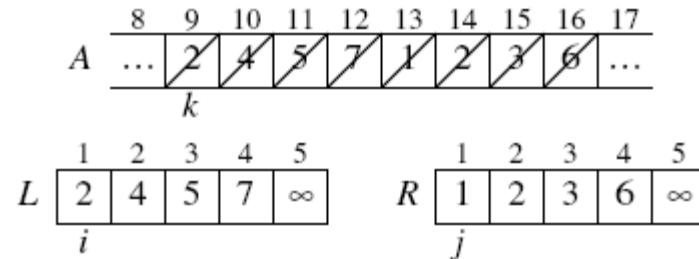
CONQUISTAR

COMBINAR

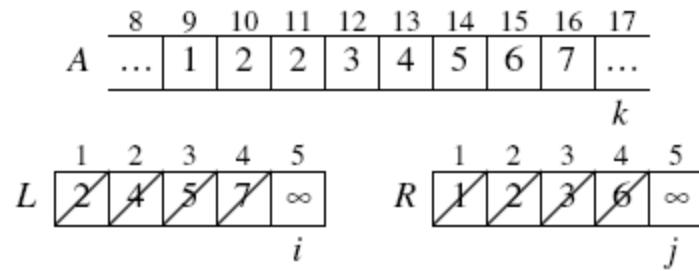
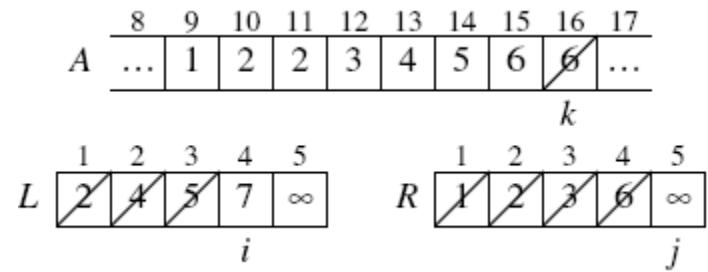
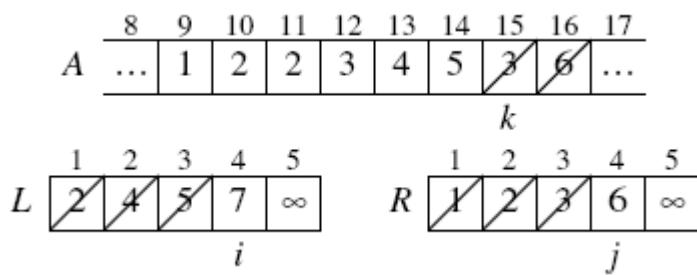
# Exemplo: Mergesort

```
MERGE (A, p, q, r)
n1 ← q-p+1
n2 ← r-q
criar arranjos L[1..n1+1] e R[1..n2+1]
for i ← 1 to n1 do
    L[i] ← A[p+i-1]
for j ← 1 to n2 do
    R[j] ← A[q+j]
L[n1+1] ← ∞
R[n2+1] ← ∞
i ← 1
j ← 1
for k ← p to r do
    if L[i] ≤ R[j] then
        A[k] ← L[i]
        i ← i+1
    else
        A[k] ← R[j]
        j ← j+1
```

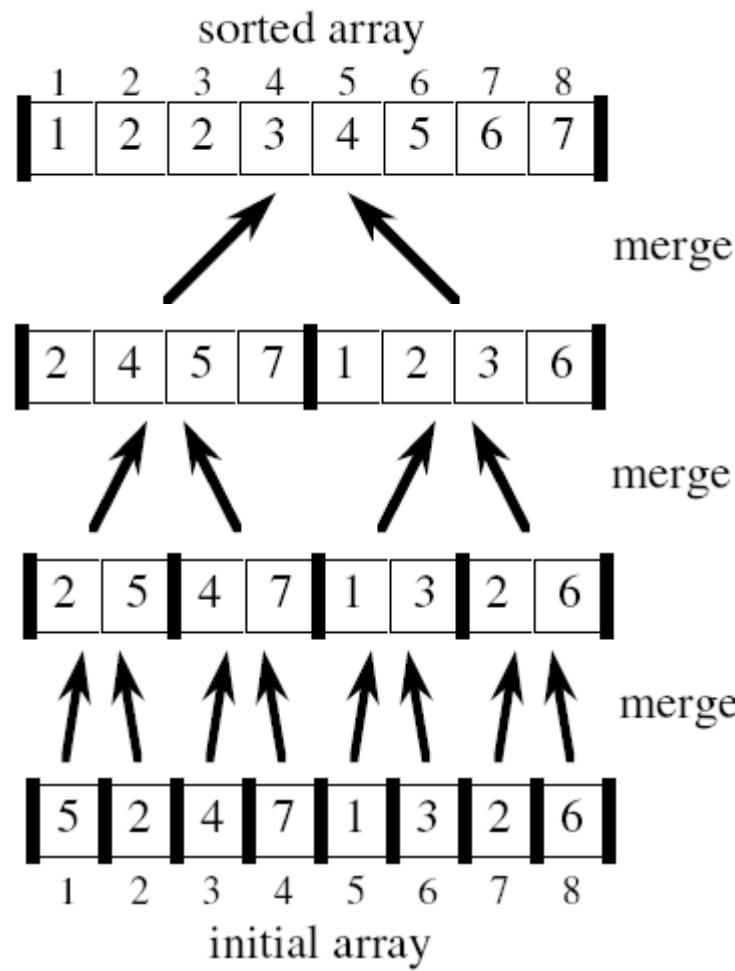
# MERGE (A,9,12,16)



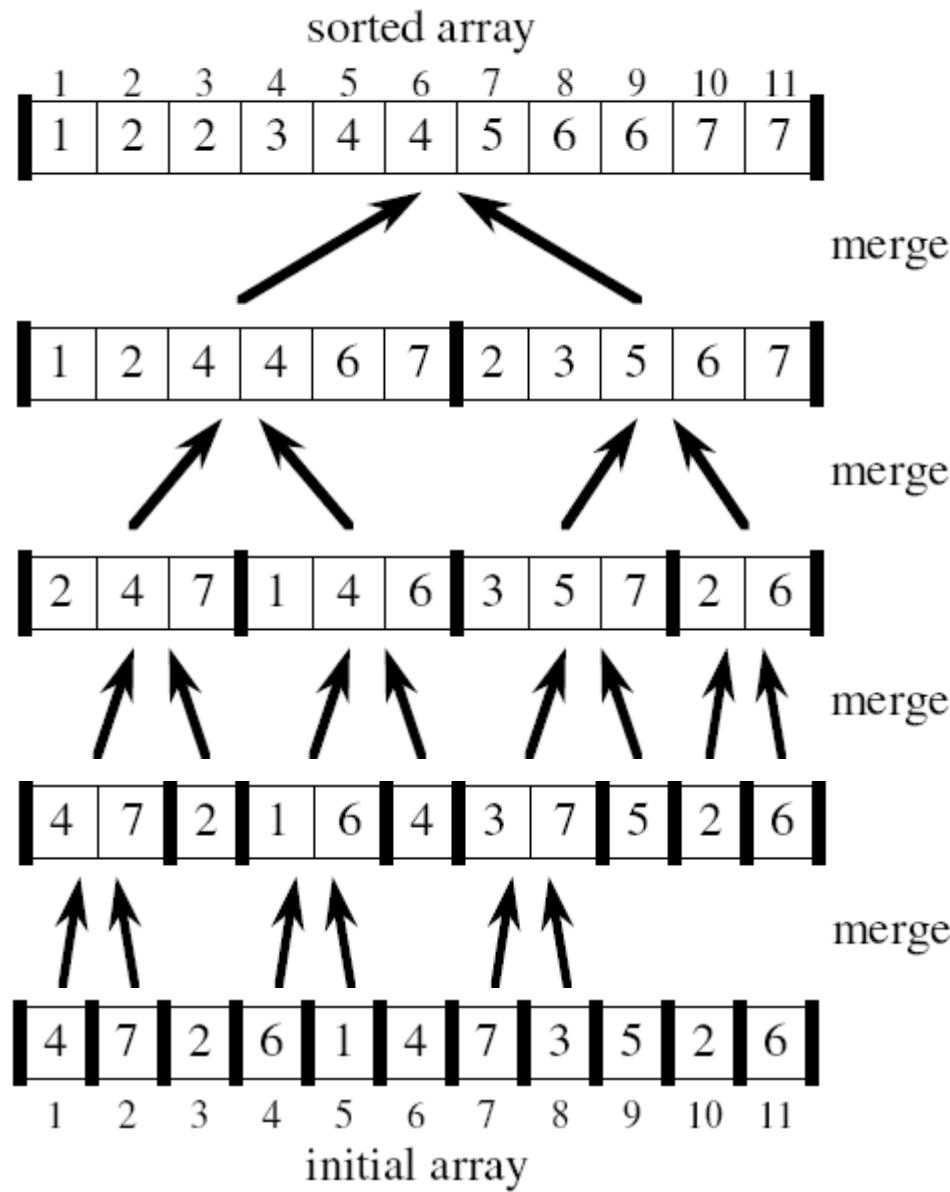
# MERGE (A,9,12,16)



# Exemplo: Mergesort



# Exemplo: Mergesort



# Análise do Mergesort

- Análise de Complexidade (supondo  $n$  par e que a operação relevante seja a comparação com os elementos do vetor):
  - **Dividir:** a etapa de dividir simplesmente calcula o ponto médio do subarranjo e não realiza comparação
  - **Conquistar:** resolvemos recursivamente dois subproblemas, cada um tem o tamanho  $n/2$  e contribui com  $2T(n/2)$  para o tempo de execução
  - **Combinar:** o procedimento MERGE leva o tempo  $n$ , onde  $n=r-p+1$  é o número de elementos que estão sendo intercalados

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

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

# Quando Utilizar Divisão e Conquista

- Existem três condições que indicam que a estratégia de divisão e conquista pode ser utilizada com sucesso:
  - Deve ser possível decompor uma instância em subinstâncias
  - A combinação dos resultados deve ser eficiente (muitas vezes, trivial)
  - As subinstâncias devem ser mais ou menos do mesmo tamanho

# Exemplo: Quicksort

```
void partition (int a[], int esq, int dir, int &i, int &j) {  
    int aux, x;  
    i = esq;  
    j = dir;  
    x = a[(i+j)/2];  
    while (i <= j) {  
        while (x > a[i]) i++;  
        while (x < a[j]) j--;  
        if (i<=j) {  
            aux = a[i];  
            a[i] = a[j];  
            a[j] = aux;  
            i++;  
            j--;  
        }  
    }  
}  
void quicksort (int a[], int esq, int dir) {  
    int i, j;  
    partition(a, esq, dir, i, j);  
    if (esq < j)  
        quicksort(a, esq, j);  
    if (i < dir)  
        quicksort(a, i, dir);  
}
```

# Exemplo: Seleção

- Encontrar o  $k$ th menor elemento de um conjunto de números

```
int partition (int a[], int esq, int dir) {  
    int i, j, aux, x;  
    i = esq;  
    j = dir;  
    x = a[(i+j)/2];  
    while (i <= j) {  
        while (x > a[i]) i++;  
        while (x < a[j]) j--;  
        if (i<=j) {  
            aux = a[i];  
            a[i] = a[j];  
            a[j] = aux;  
            i++;  
            j--;  
        }  
    }  
    return (i-1);  
}
```

```
int selection(int a[], int l, int r, int k) {  
    int i;  
    if (r > l) {  
        i = partition(a, l, r);  
        if (i == k)  
            return (i);  
        if (i > l+k-1)  
            return (selection(a, l, i-1, k));  
        if (i < l+k-1)  
            return (selection(a, i+1, r, k-i));  
    }  
}
```

# Exercício

3. Implemente a função merge com custo de  $n-1$  comparações no pior caso. Encontre a função de complexidade do Mergesort quando o mesmo utiliza essa função merge.
  
4. Faça a análise de complexidade do melhor caso do Quicksort e do algoritmo para encontrar o  $k^{th}$  menor elemento de um conjunto de números

# Programação Dinâmica

Ziviani – págs. 54 até 57

Cormen – págs. 259 até 295

# Programação Dinâmica

- **Divisão e Conquista**
  - Problema é partido em subproblemas que se resolvem separadamente
  - A solução é obtida por combinação das soluções
  - É *top-down*
- **Programação Dinâmica**
  - Resolvem-se os problemas de pequena dimensão e guardam-se as soluções
  - A solução de um problema é obtida combinando as de problemas de menor dimensão
  - É *bottom-up*
  - Calcula a solução para todos os subproblemas, partindo dos subproblemas menores para os maiores, armazenando os resultados em uma tabela
  - A vantagem é que uma vez que um subproblema é resolvido, a resposta é armazenada em uma tabela e nunca mais é recalculado

# Aplicação Direta – Números de Fibonacci

- Encontrar o  $n$ -ésimo número de Fibonacci

$$f_0 = 0, f_1 = 1,$$

$$f_n = f_{n-1} + f_{n-2} \text{ para } n \geq 2$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

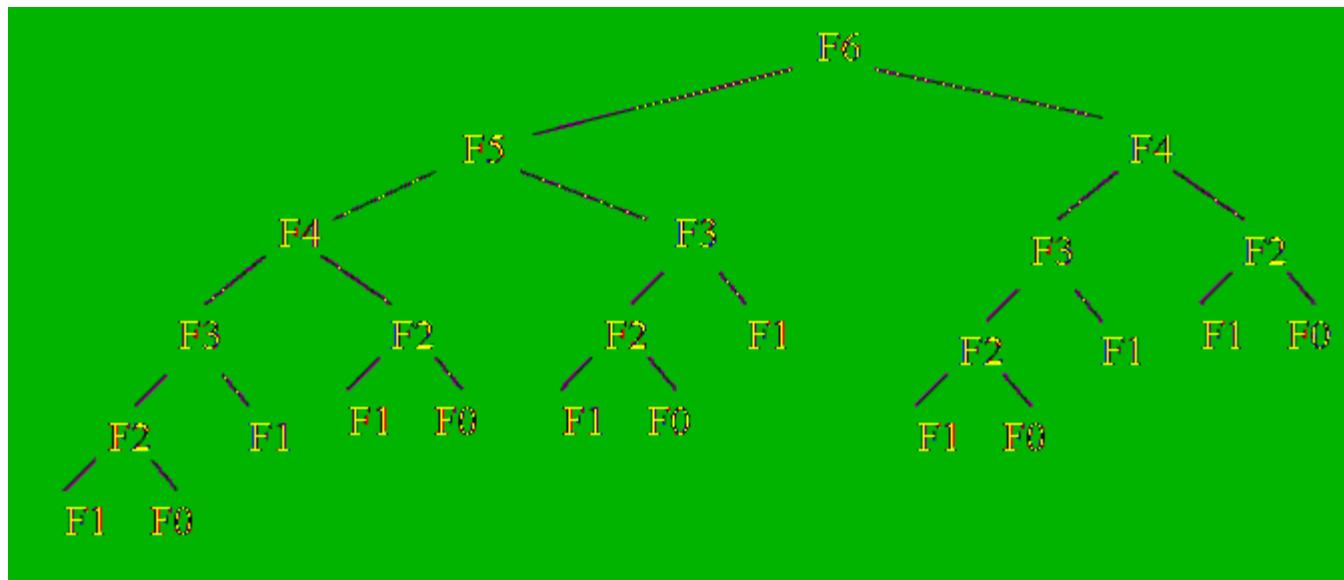
```
unsigned int FibRec (unsigned int n) {  
    if (n < 2)  
        return n;  
    else  
        return (FibRec(n-1) + FibRec(n-2));  
}
```

Simples!!! Elegante!!!

Mas .... vamos analisar o seu desempenho!!!

# Aplicação Direta – Números de Fibonacci

- Problema do algoritmo recursivo: repetição de chamadas iguais



# Aplicação Direta – Números de Fibonacci

- Análise de Complexidade:

- Considerando que a medida de complexidade de tempo  $T(n)$  é o número de adições, então

$$\begin{cases} T(n) = T(n-1) + T(n-2) + 1 & \text{para } n \geq 2 \\ T(n) = 0 & \text{para } n \leq 1 \end{cases}$$

- $T(n) = \Theta(1,618^n)$

Algoritmo recursivo é exponencial

Problema: repetição de chamadas

# Aplicação Direta – Números de Fibonacci

- Solução iterativa

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

Resolvemos  
problemas  
menores

Em todo estágio, guardamos  
as soluções para os dois  
problemas anteriores nas  
variáveis  $i$  e  $F$

- Considerando que a medida de complexidade de tempo  $T(n)$  é o número de adições, então  $T(n) = \Theta(n)$

Devemos evitar o uso de recursividade quando  
existe solução óbvia por iteração

# Programação Dinâmica

- Abordagem
  - Resolva problemas menores
  - Armazene as soluções
  - Use aquelas soluções para resolver problemas maiores
- Algoritmos dinâmicos usam espaço!

# Problema da Mochila

- O ladrão encontra o cofre cheio de itens de vários tamanhos e valores, mas tem apenas uma mochila de capacidade limitada. Qual é a combinação de itens que deve levar para maximizar o valor do roubo?

Exemplo	
Tamanho:	3
Valor:	4
Nome:	A
Tamanho:	4
Valor:	5
Nome:	B
Tamanho:	7
Valor:	10
Nome:	C
Tamanho:	8
Valor:	11
Nome:	D
Tamanho:	9
Valor:	13
Nome:	E

- Considerando que a mochila tem capacidade 17, qual é a melhor combinação?

# Problema da Mochila

- Muitas situações de interesse comercial
  - Melhor forma de carregar um caminhão ou avião
- Tem variantes: número de itens de cada tipo pode ser limitado
- Abordagem programação dinâmica:
  - Calcular a melhor combinação para todas as mochilas de tamanho até  $M$
  - Cálculo é eficiente se feito na ordem apropriada

# Problema da Mochila

```
for( j = 1; j <= N; j++ )
{
    for( i=1; i <= M; i++ )
        if ( i >= size[j] )
            if ( cost[i] < cost[i-size[j]] + val[j] )
                {
                    cost[i] = cost[i-size[j]] + val[j];
                    best[i] = j;
                }
}
```

- `cost[i]`: maior valor que se consegue com mochila de capacidade  $i$
- `best[i]`: último item acrescentado para obter o máximo
- Calcula-se o melhor valor que se pode obter usando só itens tipo A, para todos os tamanhos de mochila
- Repete-se usando só A's e B's, e assim sucessivamente

# Problema da Mochila

- Quando um item  $j$  é escolhido para a mochila: o melhor valor que se pode obter é  $\text{val}[j]$  (do item) mais  $\text{cost}[i-\text{size}[j]]$  (para encher o resto)
- Se o valor assim obtido é superior ao que se consegue sem usar o item  $j$ , atualiza-se  $\text{cost}[i]$  e  $\text{best}[i]$ ; senão mantém-se
- Conteúdo da mochila ótima: recuperado através do vetor  $\text{best}[i]$ 
  - $\text{best}[i]$  indica o último item da mochila
  - O restante é o indicado para a mochila de tamanho  $M-\text{size}[\text{best}[M]]$
- Eficiência: A solução em programação dinâmica gasta tempo  $\Theta(NM)$

# Problema da Mochila

	k	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j=1	cost[k]	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
	best[k]	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
j=2	cost[k]	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
	best[k]	A	B	B	A	B	B	A	B	B	A	B	B	A	B	B
j=3	cost[k]	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
	best[k]	A	B	B	A	C	B	A	C	C	A	C	C	A	C	C
j=4	cost[k]	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
	best[k]	A	B	B	A	C	D	A	C	C	A	C	C	D	C	C
j=5	cost[k]	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
	best[k]	A	B	B	A	C	D	E	C	C	E	C	C	D	E	C

# Exercício

5. Considere os seguintes itens e que a mochila tem capacidade 20, qual é a combinação de itens que devemos levar para maximizar o valor?

Item	A	B	C	D
Tamanho	3	4	5	6
Valor	13	15	20	15

# Multiplicação de uma Cadeia de Matrizes

- Dada uma sequência de matrizes de dimensões diversas, como fazer o seu produto minimizando o esforço computacional
- Exemplo:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} \begin{bmatrix} c_{11} \\ c_{21} \\ c_{31} \end{bmatrix} \begin{bmatrix} d_{11} & d_{12} \\ e_{21} & e_{22} \end{bmatrix} \begin{bmatrix} e_{11} & e_{12} \\ f_{21} & f_{22} \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \end{bmatrix}$$

- Multiplicando da esquerda para a direita: 84 operações
- Multiplicando da direita para a esquerda: 69 operações
- **Qual a melhor sequência?**

# Multiplicação de uma Cadeia de Matrizes

- Multiplicar  $N$  matrizes  $M_1 M_2 M_3 \dots M_N$  na qual  $M_i$  tem  $r_i$  linhas e  $r_{i+1}$  colunas
- Multiplicação de uma matriz  $p \times q$  por outra  $q \times r$  produz uma matriz  $p \times r$  requerendo  $q$  produtos para cada entrada, totalizando  $pqr$  operações de multiplicação
- Algoritmo em programação dinâmica:
  - Multiplicar 2 matrizes: só há uma maneira de multiplicar; registra-se o custo
  - Multiplicar 3 matrizes: o menor custo de realizar  $M_1 M_2 M_3$  é calculado comparando os custos de multiplicar  $M_1 M_2$  por  $M_3$  e de multiplicar  $M_1$  por  $M_2 M_3$ ; registra-se o menor custo
  - O procedimento repete-se para sequências de tamanho crescente

# Multiplicação de uma Cadeia de Matrizes

- No geral:
  - Para  $1 \leq j \leq N-1$  encontra-se o custo mínimo de calcular  $M_i M_{i+1} \dots M_{i+j}$  encontrando, para  $1 \leq i \leq N-j$  e para cada  $k$  entre  $i$  e  $i+j$  os custos de obter  $M_i M_{i+1} \dots M_{k-1}$  e  $M_k M_{k+1} \dots M_{i+j}$  somando o custo de multiplicar esses resultados

# Multiplicação de uma Cadeia de Matrizes

```
for( i=1; i <= N; i++ )  
    for( j = i+1; j <= N; j++ ) cost[i][j] = INT_MAX;  
for( i=1; i <= N; i++ ) cost[i][i] = 0;  
for( j=1; j < N; j++ )  
    for( i=1; i <= N-j; i++ )  
        for( k= i+1; k <= i+j; k++ )  
        {  
            t = cost[i][k-1] + cost[k][i+j] +  
                r[i]*r[k]*r[i+j+1];  
            if( t < cost[i][i+j] )  
                { cost[i][i+j] = t; best[i][i+j] = k;  
                }  
        }  
    }
```

# Multiplicação de uma Cadeia de Matrizes

- Para  $1 \leq j \leq N-1$  encontra-se o custo mínimo de calcular  $M_i M_{i+1} \dots M_{i+j}$ 
  - Para  $1 \leq i \leq N-j$  e para cada  $k$  entre  $i$  e  $i+j$  calculam-se os custos para obter  $M_i M_{i+1} \dots M_{k-1}$  e  $M_k M_{k+1} \dots M_{i+j}$
  - Soma-se o custo de multiplicar esses 2 resultados
- Cada grupo é partido em grupos menores
  - Custos mínimos para os 2 grupos são vistos numa tabela
- Custo da multiplicação final  $M_i M_{i+1} \dots M_{k-1}$  é uma matriz  $r_i \times r_k$  e  $M_k M_{k+1} \dots M_{i+j}$  é uma matriz  $r_k \times r_{i+j+1}$ , o custo de multiplicar as duas é  $r_i r_k r_{i+j+k}$

# Multiplicação de uma Cadeia de Matrizes

- $\text{cost}[l][r]$  é o custo mínimo para  $M_l M_{l+1} \dots M_r$
- Programa obtém  $\text{cost}[i][i+j]$  para  $1 \leq i \leq N-j$  com  $j$  de 1 a  $N-1$
- Chegando a  $j=N-1$ , tem-se o custo de calcular  $M_1 M_2 \dots M_N$
- Recuperar a sequência ótima
  - Guarda o registro das decisões feitas para cada dimensão
  - Permite recuperar a sequência de custo mínimo

# Multiplicação de uma Cadeia de Matrizes

	B	C	D	E	F
A	24 [A][B]	14 [A][BC]	22 [ABC][D]	26 [ABC][DE]	36 [ABC][DEF]
B		6 [B][C]	10 [BC][D]	14 [BC][DE]	22 [BC][DEF]
C			6 [C][D]	10 [C][DE]	19 [C][DEF]
D				4 [D][E]	10 [DE][F]
E					12 [E][F]

# Exercícios

6. Encontre uma colocação ótima de parênteses de um produto de cadeias de matrizes cuja sequência de dimensões é  $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$

# Árvore de Pesquisa Binária Ótima

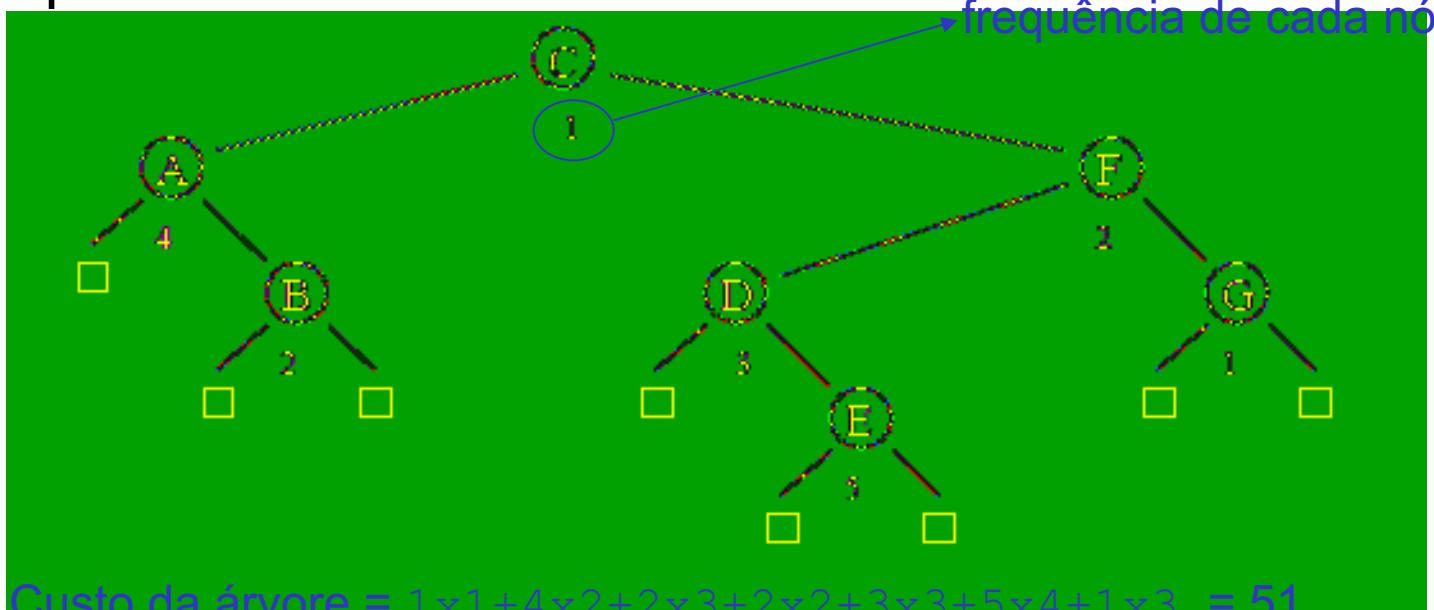
- Em pesquisa, as chaves ocorrem com frequências diversas; exemplos:
  - Verificador ortográfico: encontra mais frequentemente as palavras mais comuns
  - Compilador de C++: encontra mais frequentemente “if” e “for” que “main”
- Usando uma árvore de pesquisa binária, é vantajoso ter mais perto do topo as chaves mais usadas
- Algoritmo de programação dinâmica pode ser usado para organizar as chaves de forma a minimizar o custo total da pesquisa

# Árvore de Pesquisa Binária Ótima

- Problema tem semelhança com o dos códigos de Huffman (minimização do tamanho do caminho externo)
  - Mas, código de Huffman não requer a manutenção da ordem das chaves
  - Na árvore de pesquisa binária, os nós à esquerda de cada nó têm chaves menores e os nos à direita têm chaves maiores
- Problema é semelhante ao da ordem de multiplicação de uma cadeia de matrizes

# Árvore de Pesquisa Binária Ótima

- Exemplo:



- Custo da árvore é o comprimento do caminho interno ponderado da árvore:
  - Multiplicar a frequência de cada nó pela sua distância à raiz
  - Soma para todos os nós

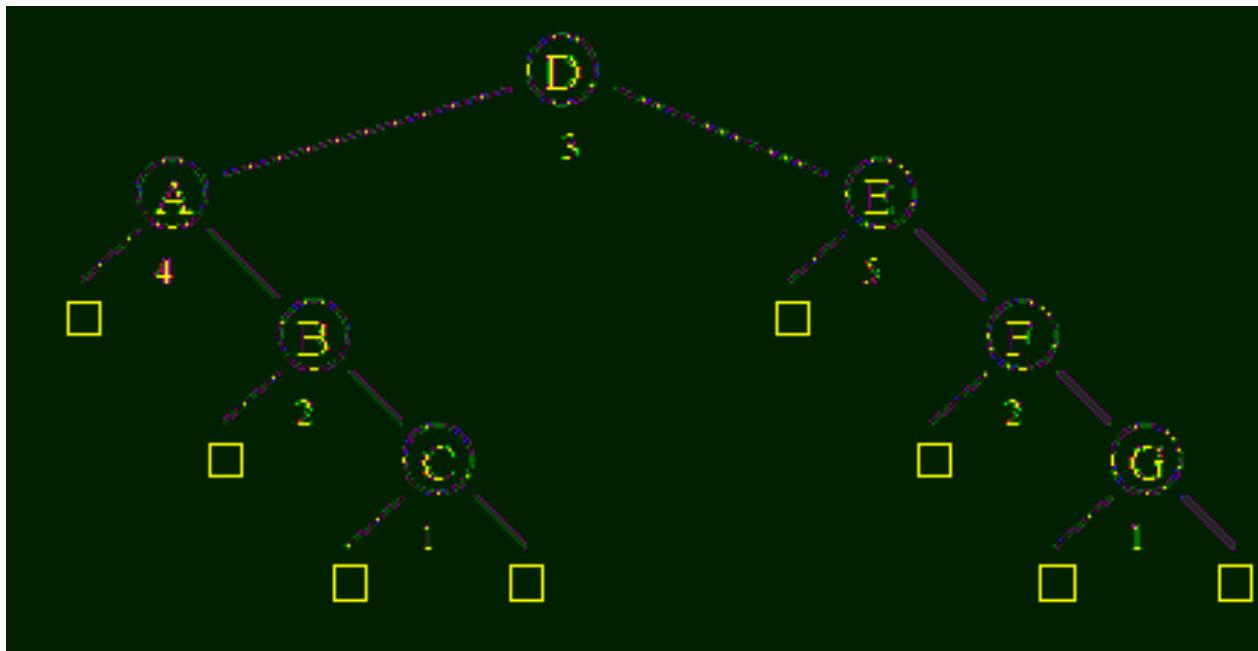
# Árvore de Pesquisa Binária Ótima

- Dados:
  - Chaves  $K_1 < K_2 < \dots < K_n$
  - Frequências  $r_1, \dots, r_n$
- Construir árvore de pesquisa que minimize a soma, para todas as chaves, dos produtos das frequências pelas distâncias à raiz
- Abordagem em programação dinâmica
  - Calcular, para cada  $j$  de 1 a  $N-1$ , a melhor maneira de construir subárvores contendo  $K_i, K_{i+1}, \dots, K_{i+j}$ , para  $1 \leq i \leq N-j$
  - Para cada  $j$ , tenta-se cada nó como raiz e usam-se os valores já computados para determinar as melhores escolhas para as subárvores
  - Para cada  $k$  entre  $i$  e  $i+j$ , construir a árvore ótima contendo  $K_i, K_{i+1}, \dots, K_{i+j}$  com  $K_k$  na raiz
  - A árvore com  $K_k$  na raiz é formada usando a árvore ótima para  $K_i, K_{i+1}, \dots, K_{k-1}$  como subárvore esquerda e a árvore ótima para  $K_{k+1}, K_{k+2}, \dots, K_{i+j}$  como subárvore direita

# Árvore de Pesquisa Binária Ótima

```
for( i=1; i <= N; i++ )
    for( j = i+1; j <= N+1; j++ ) cost[i][j] = INT_MAX;
for( i=1; i <= N; i++ ) cost[i][i] = f[i];
for( i=1; i <= N+1; i++ ) cost[i][i-1] = 0;
for( j=1; j <= N-1; j++ )
    for( i=1; i <= N-j; i++ )
    {
        for( k= i; k <= i+j; k++ )
        {
            t = cost[i][k-1] + cost[k+1][i+j];
            if( t < cost[i][i+j] )
                { cost[i][i+j] = t; best[i][i+j] = k; }
        }
        for( k= i; k <= i+j; cost[i][i+j] += f[k++]) ;
    }
```

# Árvore de Pesquisa Binária Ótima



$$\text{Custo da árvore} = 3 \times 1 + 4 \times 2 + 2 \times 3 + 1 \times 4 + 5 \times 2 + 2 \times 3 + 1 \times 4 = 41$$

# Árvore de Pesquisa Binária Ótima

- O método para determinar uma árvore de pesquisa binária ótima em programação dinâmica gasta tempo  $\Theta(N^3)$  e espaço  $\Theta(N^2)$
- Examinando o código:
  - O algoritmo trabalha com uma matriz de dimensão  $N^2$  e gasta tempo proporcional a  $N$  em cada entrada
- É possível melhorar:
  - Usando o fato de que a posição ótima para a raiz da árvore não pode ser muito distante da posição ótima para uma árvore um pouco menor, no programa dado  $k$  não precisa de cobrir todos os valores de  $i$  a  $i+j$

# Programação Dinâmica - Resumo

- Tradução iterativa inteligente da recursão
  - Resolvem problemas menores
  - Armazenam as soluções para estes problemas menores numa tabela
  - Usam as soluções dos problemas menores para obterem a solução de problemas maiores
- Cada instância do problema é resolvida a partir da solução de subinstâncias da instância original
- O problema deve ter estrutura recursiva: a solução de toda instância do problema deve “conter” soluções de subinstâncias da instância

# Exercícios

7. Determine o custo e a estrutura de uma árvore de pesquisa binária ótima para um conjunto de  $n=7$  chaves com seguinte frequência de ocorrência:

	A	B	C	D	E	F	G
$f$	7	4	5	1	4	8	7

# **Algoritmos Gulosos**

Ziviani – págs. 58 até 59

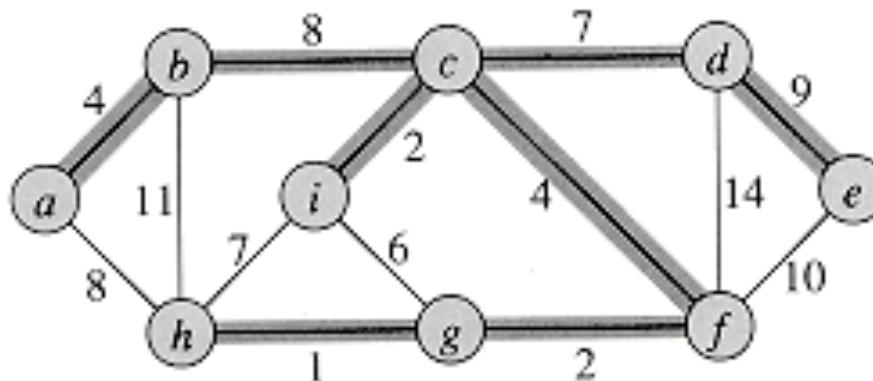
Cormen – págs. 296 até 323

# Algoritmos Gulosos

- Para resolver um problema, um algoritmo guloso escolhe, em cada iteração, a melhor opção para o momento
- A opção escolhida passa a fazer parte da solução que o algoritmo constrói
- O algoritmo faz uma escolha ótima local esperando que esta o leve a uma solução ótima global
- Um algoritmo guloso jamais se arrepende ou volta atrás, as escolhas que faz em cada iteração são definitivas

# Árvore Geradora Mínima

- Árvore Geradora Mínima é a árvore geradora de menor peso de  $G$
- Dado um grafo  $G$  com pesos associados às arestas, encontrar uma árvore geradora mínima de  $G$



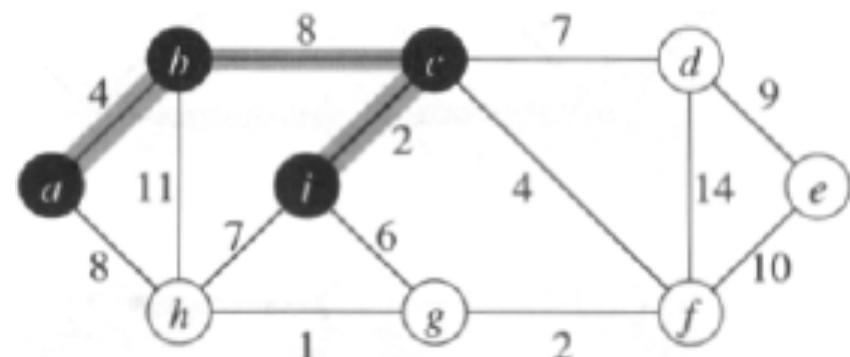
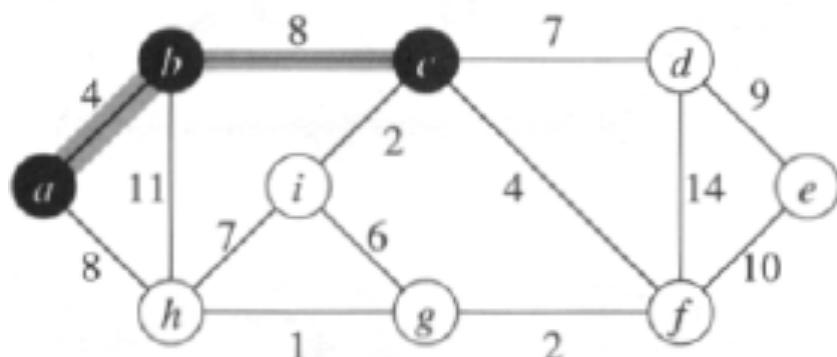
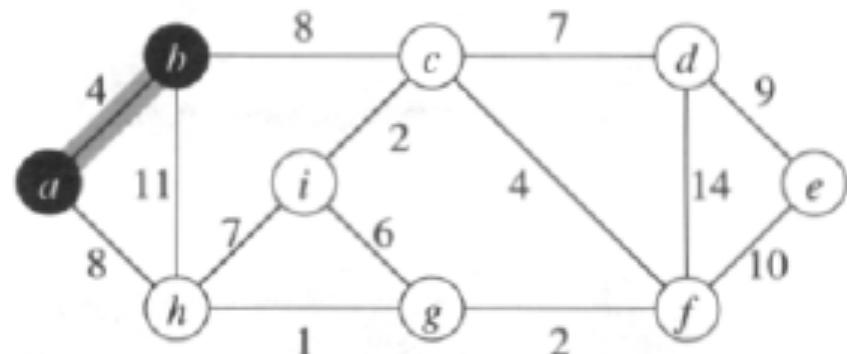
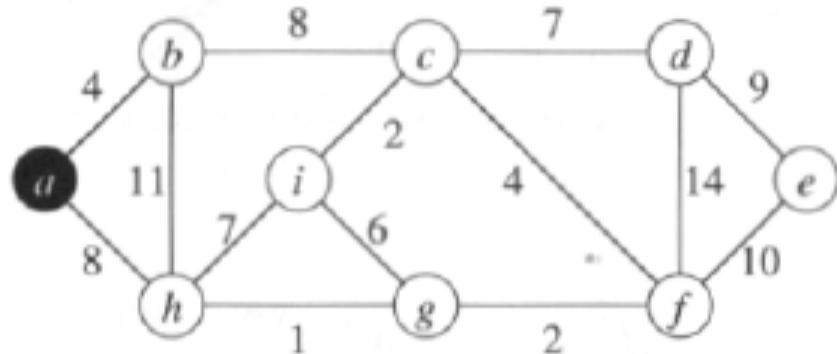
# Algoritmo de Prim

- As arestas no conjunto  $A$  sempre formam uma árvore única
- A árvore começa a partir de um vértice de raiz arbitrária  $r$  e aumenta até a árvore alcançar todos os vértices em  $V$
- Em cada etapa, uma aresta leve conectando um vértice de  $A$  a um vértice em  $V-A$  é adicionada à árvore
- Quando o algoritmo termina, as arestas em  $A$  formam uma árvore geradora mínima
- Durante a execução do algoritmo, todos os vértices que não estão na árvore residem em uma fila de prioridade mínima  $Q$  baseada em um campo chave
- Para cada vértice  $v$ , chave [ $v$ ] é o peso mínimo de qualquer aresta que conecta  $v$  a um vértice na árvore
- $\pi[v]$  é o pai de  $v$  na árvore

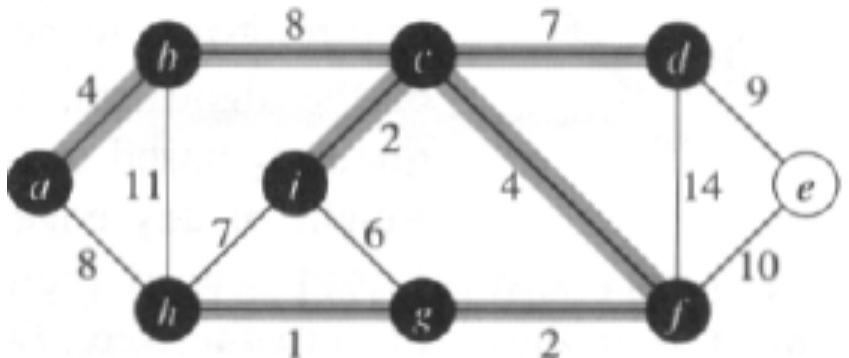
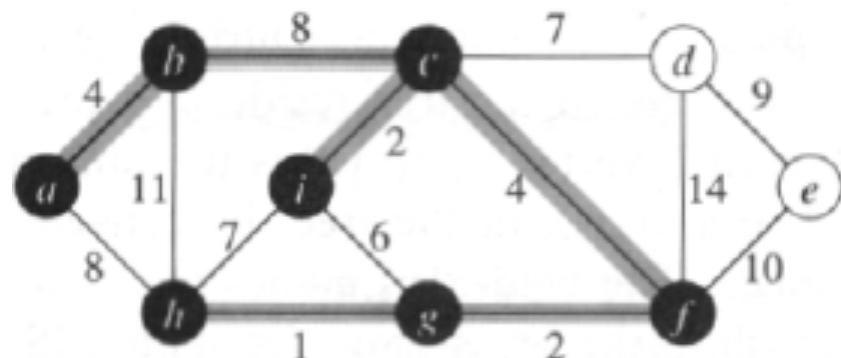
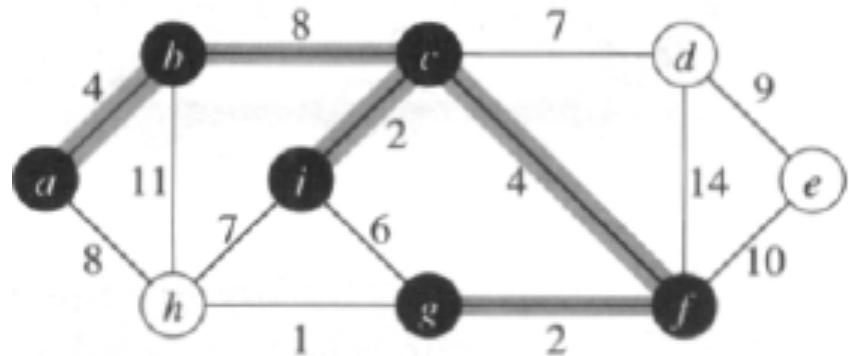
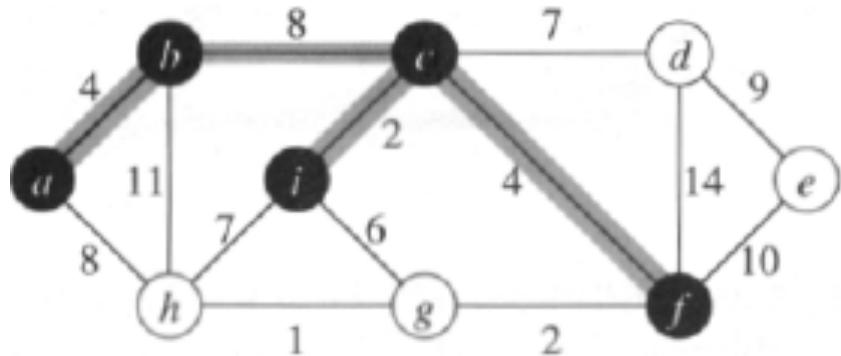
# Algoritmo de Prim

```
MST-PRIM (G, w, r)
for cada  $u \in V[G]$  do
     $chave[u] \leftarrow \infty$ 
     $\pi[u] \leftarrow \text{NIL}$ 
 $chave[r] \leftarrow 0$ 
 $Q \leftarrow V[G]$ 
while  $Q \neq 0$  do
     $u \leftarrow \text{EXTRACT-MIN}(Q)$   INSERE NA AGM
    for cada  $v \in \text{Adj}[u]$  do
        if  $v \in Q$  e  $w(u, v) < \text{chave}[v]$  then
             $\pi[v] \leftarrow u$ 
             $\text{chave}[v] \leftarrow w(u, v)$ 
```

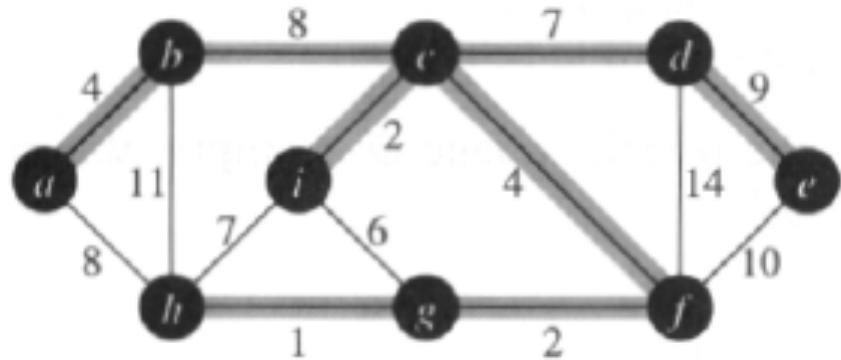
# Algoritmo de Prim



# Algoritmo de Prim



# Algoritmo de Prim



# Versões do Problema da Mochila

- Problema da Mochila 0-1 ou 0-1 *Knapsack Problem*:
  - O item  $i$  é levado integralmente ou é deixado
- Problema da Mochila Fracionário:
  - Fração do item  $i$  pode ser levada

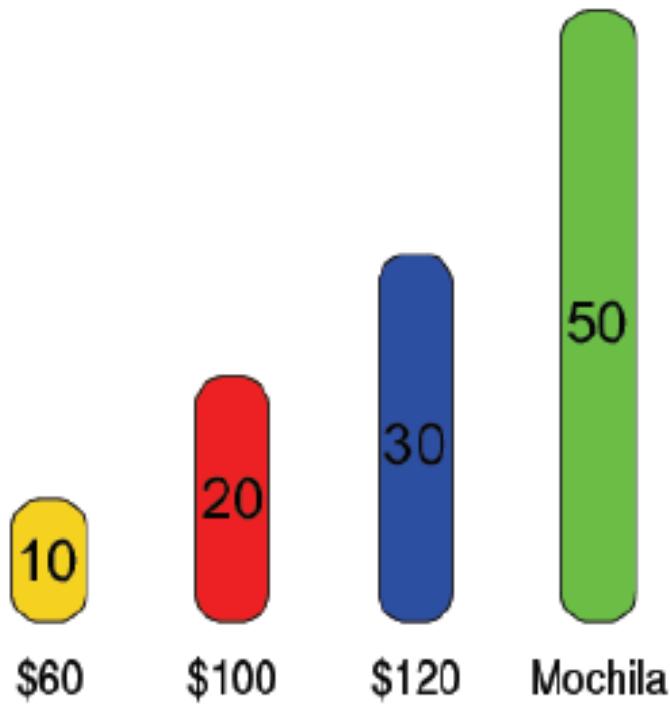
# Considerações sobre as duas versões

- Possuem a propriedade de subestrutura ótima
- Problema inteiro:
  - Considere uma carga que pesa no máximo  $W$  com  $n$  itens
  - Remova o item  $j$  da carga
  - Carga restante deve ser a mais valiosa pesando no máximo  $W - w_j$  com  $n - 1$  itens
- Problema fracionário:
  - Considere uma carga que pesa no máximo  $W$  com  $n$  itens
  - Remova um peso  $w$  do item  $j$  da carga
  - Carga restante deve ser a mais valiosa pesando no máximo  $W - w$  com  $n - 1$  itens mais o peso  $w_j - w$  do item  $j$

# Considerações sobre as duas versões

- Problema inteiro
  - Não é resolvido usando a técnica gulosa
- Problema fracionário
  - É resolvido usando a técnica gulosa
- Estratégia para resolver o problema fracionário:
  - Calcule o valor por unidade de peso  $v_i / w_i$  para cada item
  - Estratégia gulosa é levar tanto quanto possível do item de maior valor por unidade de peso
  - Repita o processo para o próximo item com esta propriedade até alcançar a carga máxima
- Complexidade para resolver o problema fracionário:
  - Ordene os itens  $i$  ( $i = 1, \dots, n$ ) pelas frações  $v_i / w_i$
  - $\Theta(n \log n)$

# Exemplo: Situação inicial Problema 0-1

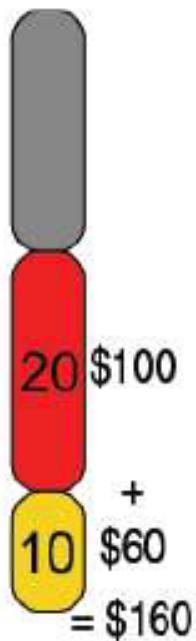
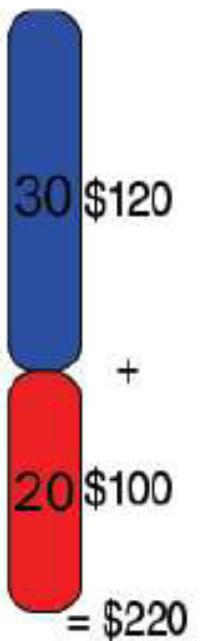


Item	Peso	Valor	V/P
1	10	60	6
2	20	100	5
3	30	120	4

Carga máxima da mochila: 50

# Exemplo: Estratégia Gulosa

## Problema 0-1



Soluções possíveis:

#	Item (Valor)
1	$2 + 3 = 100 + 120 = 220$
2	$1 + 2 = 60 + 100 = 160$
3	$1 + 3 = 60 + 120 = 180$

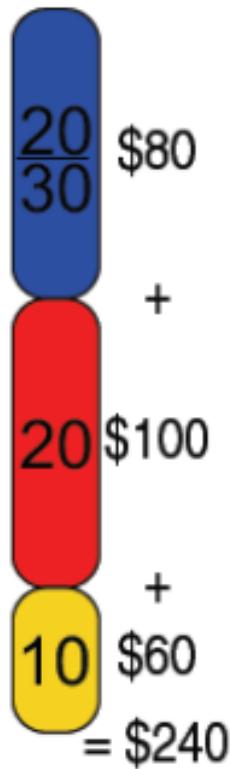
→ Solução 2 é a gulosa.

# Exemplo: Estratégia Gulosa

## Problema 0-1

- Considerações:
  - Levar o item 1 faz com que a mochila fique com espaço vazio
  - Espaço vazio diminui o valor efetivo da relação  $v/w$
  - Neste caso deve-se comparar a solução do subproblema quando:
    - ▶ Item é incluído na solução X Item é excluído da solução
  - Passam a existir vários subproblemas
  - Programação dinâmica passa a ser a técnica adequada

# Exemplo: Estratégia Gulosa Problema Fracionário



Item	Peso	Valor	Fração
1	10	60	1
2	20	100	1
3	30	80	2/3

# O Problema da Mochila Fracionária

- O algoritmo é guloso porque, em cada iteração, abocaña o objeto de maior valor específico dentre os disponíveis, sem se preocupar com o que vai acontecer depois. O algoritmo jamais se arrepende do valor atribuído a um componente de  $x$

# Exercício

8. Resolva o problema da mochila fracionária considerando uma mochila com capacidade 50 e 4 itens conforme peso e valor especificados na tabela abaixo

	A	B	C	D	E
Peso	40	30	20	10	20
Valor	840	600	400	100	300

# **Teoria da Complexidade**

# Introdução

- A maioria dos problemas conhecidos e estudados se divide em dois grupos:
  - Problemas cuja solução é limitada por um polinômio de grau pequeno
    - ▶ Pesquisa binária:  $\Theta(\log n)$
    - ▶ Ordenação:  $\Theta(n \log n)$
    - ▶ Multiplicação de matriz:  $\Theta(n^{2.81})$
  - Problemas cujo melhor algoritmo conhecido é não-polinomial
    - ▶ Problema do Caixeiro Viajante:  $\Theta(n^2 2^n)$
    - ▶ Knapsack Problem:  $\Theta(2^{n/2})$

# Algoritmos Polinomiais X Algoritmos Exponenciais

- Algoritmos polinomiais são obtidos através de um entendimento mais profundo da estrutura do problema
- Um problema é considerado **tratável** quando existe um algoritmo polinomial para resolve-lo
- Algoritmos exponenciais são, em geral, simples variação de pesquisa exaustiva
- Um problema é considerado **intratável** se ele é tão difícil que não existe um algoritmo polinomial para resolve-lo

# Algoritmos Polinomiais X Algoritmos Exponenciais

- Entretanto,
  - Um algoritmo  $\Theta(2^n)$  é mais rápido que um algoritmo  $\Theta(n^5)$  para  $n \leq 20$
  - Existem algoritmos exponenciais que são muito úteis na prática
    - ▶ Algoritmo Simplex para programação linear é exponencial mas, executa muito rápido na prática
  - Na prática os algoritmos polinomiais tendem a ter grau 2 ou 3 no máximo e não possuem coeficientes muito grandes  $n^{100}$  ou  $10^{99}n^2$  NÃO OCORREM

# Decisão x Otimização

- Em um problema de **otimização** queremos determinar uma solução possível com o melhor valor.
- Em um problema de **decisão** queremos responder “sim” ou “não”.
- Para cada problema de otimização podemos encontrar um problema de decisão equivalente a ele.

# Problemas “Sim/Não” ou Problemas de Decisão

- Para o estudo teórico da complexidade de algoritmos é conveniente considerar problemas cujo resultado seja “sim” ou “não”
- Exemplo: Problema do Caixeiro Viajante
  - **Dados:** Um conjunto de cidades  $C = \{c_1, c_2, \dots, c_n\}$ , uma distância  $d(c_i, c_j)$  para cada par de cidades  $c_i, c_j \in C$  e uma constante  $K$ .
  - **Questão:** Existe um roteiro para todas as cidades em  $C$  cujo comprimento total seja menor ou igual a  $K$ ?

# Classe P e NP

- Classe P:
  - Um algoritmo está na Classe P se a complexidade do seu pior caso é uma função polinomial do tamanho da entrada de dados
- Classe NP:
  - Classe de problemas “Sim/Não” para os quais uma dada solução pode ser verificada facilmente
  - Existe uma enorme quantidade de problemas em NP para os quais não se conhece um único algoritmo polinomial para resolver qualquer um deles

# Ordenação está em NP

```
VOrdenacao(A, n)
```

```
inicio
```

```
    ordenado ← verdadeiro
```

```
    para i ← 1 até n-1 faça
```

```
        se A[i] > A[i+1] então
```

```
            ordenado ← falso
```

```
        fim se
```

```
    fim para
```

```
    se ordenado = falso então
```

```
        escreva "NAO"
```

```
    senão
```

```
        escreva "SIM"
```

```
    fim se
```

```
fim
```

- Complexidade:  $\Theta(n)$

# Coloração de Grafos está em NP

```
VColoracao(G, C, K)
inicio
    colorido ← verdadeiro
    para i ← 1 até |E| faça
        se C[Ei.V1] = C[Ei.V2] então
            colorido ← falso
        fim se
    fim para
    se colorido = verdadeiro e |C| ≤ K então
        escreva "SIM"
    senão
        escreva "NAO"
    fim se
fim
```

- Complexidade:  $\Theta(n^2)$ , onde n é o número de vértices

# Algoritmos Não-deterministas

- Um computador não-determinista, quando diante de duas ou mais alternativas, é capaz de produzir cópias de si mesmo e continuar a computação independentemente para cada alternativa
- Um algoritmo não-determinista é capaz de escolher uma dentre as várias alternativas possíveis a cada passo (o algoritmo é capaz de adivinhar a alternativa que leva a solução)

# Algoritmos Não-deterministas

- Utilizam
  - a função escolhe ( $C$ ) : escolhe um dos elementos de  $C$  de forma arbitrária.
  - SUCESSO: sinaliza uma computação com sucesso
  - INSUCESSO: sinaliza uma computação sem sucesso
- Sempre que existir um conjunto de opções que levam a um término com sucesso então, este conjunto é sempre escolhido
- A complexidade da função escolhe é  $\Theta(1)$

# Exemplo: Pesquisa

- Pesquisar o elemento  $x$  em um conjunto de elementos  $A[1..n]$ ,  $n \geq 1$

```
j ← escolhe(A, x)
se A[j] = x então
    SUCESSO
senão
    INSUCESSO
fim se
```

- Complexidade:  $\Theta(1)$
- Para um algoritmo determinista a complexidade é  $\Theta(n)$

# Exemplo: Ordenação

- Ordenar um conjunto  $A$  contendo  $n$  inteiros  $n \geq 1$

```
NDOrdenacao (A, n)
  inicio
    para i←1 até n faça B[i]=0;
    para i←1 até n faça
      inicio
        j ← escolhe (A, i);
        se B[j] = 0 então
          B[j] = A[i];
        senão
          INSUCESSO
        fim se
      fim para
      SUCESSO
    fim
```

- $B$  contém o conjunto ordenado
- A posição correta em  $B$  de cada inteiro de  $A$  é obtida de forma não-determinista

- Complexidade do algoritmo não-determinista:  $\Theta(n)$
- Complexidade do algoritmo determinista:  $\Theta(n \log n)$

# Algoritmos Deterministas X Algoritmos Não-deterministas

- Classe P (*Polynomial-time Algorithms*)
  - Conjunto de todos os problemas que podem ser resolvidos por algoritmos deterministas em tempo polinomial
- Classe NP (*Nondeterministic Polynomial Time Algorithms*)
  - Conjunto de todos os problemas que podem ser resolvidos por algoritmos não-deterministas em tempo polinomial

# Como Mostrar que um Determinado Problema está em NP?

- Basta apresentar um algoritmo não-determinista que execute em tempo polinomial para resolver o problema

OU

- Basta encontrar um algoritmo determinista polinomial para verificar que uma dada solução é válida

# Caixeiro Viajante está em NP

- Algoritmo não-determinista em tempo polinomial

```
NDPCV(G, n, k)
inicio
    Soma ← 0
    para i ← 1 até n faça
        A[i] ← escolhe(G, n)
    fim para
    A[n+1] ← A[1]
    para i ← 1 até n faça
        Soma ← Soma + distancia entre A[i] e A[i+1]
    fim para
    se Soma ≤ k então
        SUCESSO
    senão
        INSUCESSO
    fim se
fim
```

- Complexidade do algoritmo não-determinista:  $\Theta(n)$
- Complexidade do algoritmo determinista:  $\Theta(n^2 \cdot 2^n)$

# Caixeiro Viajante está em NP

- Algoritmo determinista polinomial para verificar a solução

```
DPCVV (G, S, n, k)
inicio
    Soma ← 0
    para i ← 1 até n faca
        Soma ← Soma + distancia entre S[i] e S[i+1]
    se Soma ≤ k então
        escreva "SIM"
    senão
        escreva "NAO"
    fim
```

- Complexidade:  $\Theta(n)$

# P = NP ou P ≠ NP ?

- Como algoritmos deterministas são apenas um caso especial de algoritmos não-deterministas, podemos concluir que

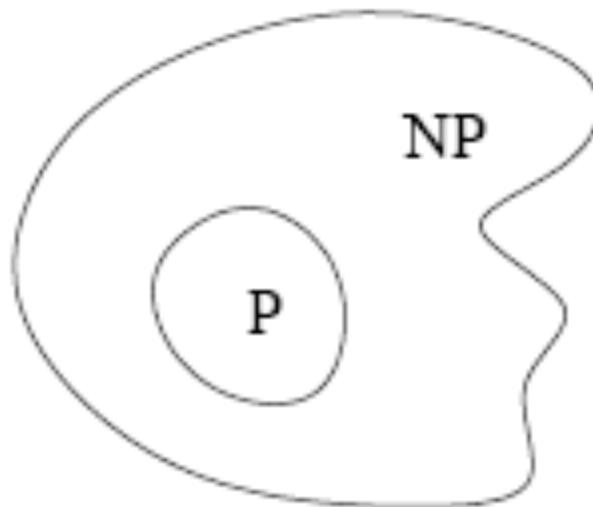
$$P \subseteq NP$$

- O que não sabemos é se

$$P = NP \text{ ou } P \neq NP$$

- Será que existem algoritmos polinomiais deterministas para todos os problemas em NP?
- Por outro lado, a prova de que  $P \neq NP$  parece exigir técnicas ainda desconhecidas

# Descrição tentativa de NP



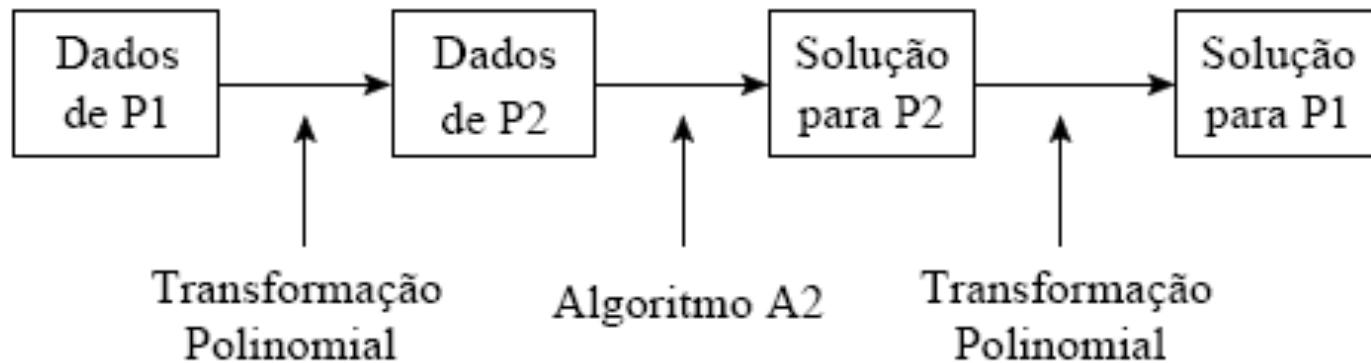
- P está contida em NP
- Acredita-se que NP seja muito maior que P

# Consequências

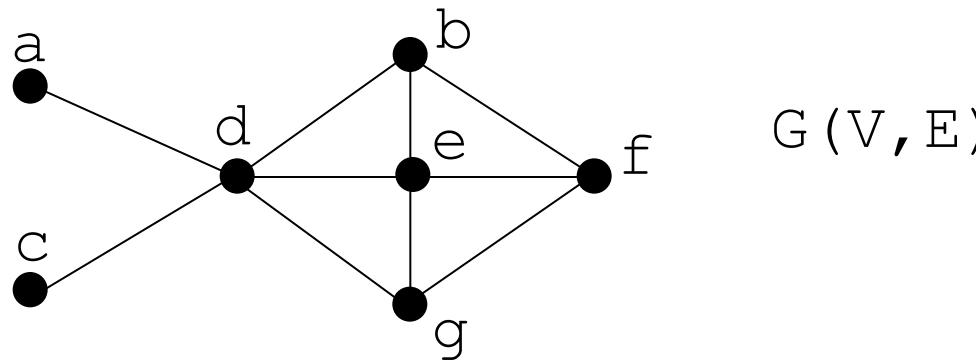
- Existem muitos problemas práticos em NP que podem ou não pertencer a P (não conhecemos nenhum algoritmo eficiente que execute em uma máquina determinista)
- Se conseguirmos provar que um problema não pertence a P, então não precisamos procurar por uma solução eficiente para ele
- Como não existe tal prova sempre há esperança de que alguém descubra um algoritmo eficiente
- Quase ninguém acredita que  $NP = P$
- Existe um esforço considerável para provar o contrário: MAS O PROBLEMA CONTINUA EM ABERTO!

# Redução Polinomial

- Sejam  $\Pi_1$  e  $\Pi_2$  dois problemas “sim/não”.
- Suponha que exista um algoritmo A2 para resolver  $\Pi_2$ . Se for possível transformar  $\Pi_1$  em  $\Pi_2$  e sendo conhecido um processo de transformar a solução de  $\Pi_2$  numa solução de  $\Pi_1$  então, o algoritmo A2 pode ser utilizado para resolver  $\Pi_1$ .
- Se estas duas transformações puderem ser realizadas em tempo polinomial então,  $\Pi_1$  é polinomialmente redutível a  $\Pi_2$  ( $\Pi_1 \leq_p \Pi_2$ )



# Exemplo de Transformação Polinomial



- Conjunto independente de vértices
  - $V' \subseteq V$  tal que todo par de vértices de  $V'$  é não adjacente, ou seja, se  $v, w \in V' \Rightarrow (v, w) \notin E$
  - a, c, b, g é um exemplo de um conjunto independente de cardinalidade 4
- Clique
  - $V' \subseteq V$  tal que todo par de vértices de  $V'$  é adjacente,  $V'$  é um subgrafo completo, ou seja, se  $v, w \in V' \Rightarrow (v, w) \in E$
  - d, b, e é um exemplo de um clique de cardinalidade 3

# Exemplo de Transformação Polinomial

- Instância  $I$  do Clique
  - **Dados:** Grafo  $G(V, E)$  e um inteiro  $k > 0$
  - **Decisão:**  $G$  possui um clique de tamanho  $\geq k$ ?
- Instância  $f(I)$  do Conjunto Independente
  - Considere o grafo complementar  $\bar{G}$  de  $G$  e o mesmo inteiro  $k$ ,  $f$  é uma transformação polinomial porque:
    1.  $\bar{G}$  pode ser obtido a partir de  $G$  em tempo polinomial
    2.  $G$  possui clique de tamanho  $\geq k$  se e somente se  $\bar{G}$  possui conjunto independente de vértices de tamanho  $\geq k$

# Exemplo de Transformação Polinomial

Se existir um algoritmo que resolva o conjunto independente em tempo polinomial, este algoritmo pode ser utilizado para resolver o clique também em tempo polinomial

**Clique  $\propto$  Conjunto Independente**

# Satisfabilidade

- Definir se uma expressão booleana  $E$  contendo produto de adições de variáveis booleanas é satisfatível
  - Exemplo:  $(x_1 + x_2) * (x_1 + \overline{x_3} + x_2) * (x_3)$   
onde  $x_i$  representa variáveis lógicas
    - + representa OR
    - \* representa AND
    - $\overline{x}$  representa NOT
- **Problema:** Existe uma atribuição de valores lógicos ( $V$  ou  $F$ ) às variáveis que torne a expressão verdadeira (“satisfaça”)?

$x_1=F, x_2=V, x_3=V$  Satisfaz!

- Exemplo:  $(x_1) * (\overline{x_1})$  não é satisfatível

# Satisfabilidade

```
NDAval(E, n)
inicio
    para i<-1 até n faça
        xi <- escolhe(true, false)
        se E(x1, x2, ..., xn) = true então
            SUCESSO
        senão
            INSUCESSO
        fim se
    fim
```

- O algoritmo obtém uma das  $2^n$  atribuições possíveis de forma não-determinista em  $O(n)$

**SAT**  $\in$  **NP**

# Teorema de Cook

- S. Cook formulou a seguinte questão (em 1971): existe algum problema em NP tal que se ele for mostrado estar em P então este fato implicaria que  $P=NP$ .
- **Teorema de Cook:** Satisfabilidade está em P se, e somente se,  $P = NP$
- Isto é: Se existe um algoritmo polinomial determinista para a satisfabilidade então, todos os problemas em NP poderiam ser resolvidos em tempo polinomial

# Teorema de Cook

- SAT está em P sse  $P = NP$
- Esboço da prova
  1. SAT está em NP. Logo, se  $P = NP$  então SAT está em P.
  2. Se SAT está em P então  $P = NP$ 
    - Prova descreve como obter de qualquer algoritmo polinomial não determinista de decisão A com entrada E uma fórmula Q(A,E) de forma que Q é satisfatível se e somente se A termina com sucesso para a entrada E. Isto significa que ele mostrou que para qualquer problema  $\Pi \in NP$ ,  $\Pi \leq_p SAT$

# NP-Completo

- Um problema de decisão  $\Pi$  é denominado NP-Completo se
  1.  $\Pi \in \text{NP}$
  2. Todo problema de decisão  $\Pi' \in \text{NP}$  satisfaz  $\Pi' \propto \Pi$
- Apenas problemas de decisão (sim/não) podem ser NP-Completo

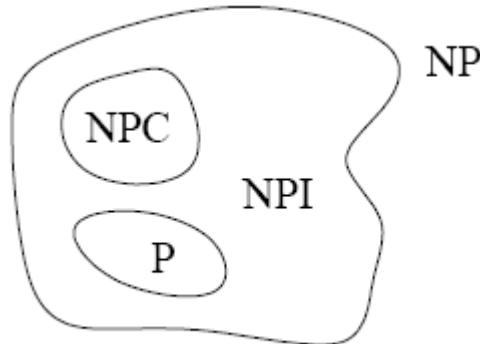
# Como Provar que um Problema é NP-Completo

1. Mostre que o problema está em NP
  2. Mostre que um problema NP-Completo conhecido pode ser polinomialmente transformado para ele
- 
- Porque:
    - Cook apresentou uma prova direta de que SAT é NP-Completo
    - Transitividade da redução polinomial

$$\text{SAT} \propto \Pi_1 \text{ e } \Pi_1 \propto \Pi_2 \Rightarrow \text{SAT} \propto \Pi_2$$

# Descrição tentativa de NP

Assumindo que  $P \neq NP$



- Se alguém encontrar um algoritmo polinomial que resolva algum problema NP-Completo então, todos os problemas em NP também terão solução polinomial, ou seja, **P será igual a NP**.
- Se alguém provar que um determinado problema em NP não tem solução polinomial então, todos os problemas em NP-Completo também não terão solução polinomial, ou seja, **P será diferente de NP**.

# Qual é a Contribuição Prática da Teoria de NP-Completo?

- Fornece um mecanismo que permite descobrir se um novo problema é “fácil” ou “difícil”
- Se encontrarmos um algoritmo eficiente para o problema, então não há dificuldades. Senão, uma prova de que o problema é NP-Completo nos diz que se acharmos um algoritmo eficiente então estaremos obtendo um grande resultado.

# Como Resolver Problemas NP-completos?

- Quando não existe solução polinomial é necessário usar **algoritmos aproximados ou heurísticas** que não garantem a solução ótima mas são rápidos

# Algoritmos Aproximados e Heurísticas

- Algoritmos aproximados:
  - Algoritmos usados normalmente para resolver problemas para os quais não se conhece uma solução polinomial
  - Devem executar em tempo polinomial dentro de limites “prováveis” de qualidade absoluta ou assintótica
- Heurísticas:
  - Algoritmos que têm como objetivo fornecer soluções sem um limite formal de qualidade, em geral avaliado empiricamente em termos de complexidade (média) e qualidade de soluções
  - É projetada para obter ganho computacional ou simplicidade conceitual, possivelmente ao custo de precisão