Análise e Complexidade de Algoritmos

Principais paradigmas do projeto de algoritmos
- Recursividade
- Tentativa e erro
- Divisão e Conquista
- Programação dinâmica
- Algoritmos Gulosos e de Aproximação

Prof. Rodrigo Rocha
prof.rodrigorocha@yahoo.com
http://www.bolinhabolinha.com

Onde Estamos

- **Ementa**
  - Revisão:
    - Estrutura de dados;Crescimento de funções;
    - Indução matemática e métodos matemáticos.
- Medidas de complexidade, análise assintótica de limites de complexidades.
- Exemplos de análise de algoritmo iterativos e recursivos.
- Análise de desempenho de alguns algoritmos clássicos de busca e ordenação.
- Introdução aos principais paradigmas do projeto de algoritmos.
- Complexidade do Problema: Limites de Complexidade, Intratabilidade, Classes P, NP, problemas Np completos e NP-difíceis.

Recursividade - Relembrando

- Definição
  - Um método/função que invoca o próprio método/função
- Atenção
  - Devemos ter um condição de parada da recursão
  - Uma condição que se torne falsa ou um contador que atinja um determinado valor
- PROBLEMA
  - Recursão que não termina

Implementação

- Um compilador implementa a recursividade através de uma pilha
- Todos os dados não globais vão para pilha
- Grava o estado corrente para poder ser recuperado
- Exemplo:
  - Algoritmo recursivo de pesquisa em árvores binárias
  - O valor do ponteiro para o nó e o endereço de retorno da chamada recursiva são armazenados na pilha

Recursão

- Vantagens
  - Código mais “limpo” e compacto
  - “Fáceis” de serem implementados
- Desvantagens
  - Consumo elevado de recursos
  - Mais difíceis de serem depurados

Exemplo

- Somatória de números
  - Ex: soma(5) = 1+2+3+4+5 = 15

```
int soma(int num)
{
    if (num==1)    /*  <====== Condicao de parada */
    {
       return num;
    }
    else {
       return num+soma(num-1); /*  <====== Chamada Recursiva */
       }
}
```

Exemplo

- Calculo do Máximo Divisor Comum (MDC)
  - Fonte: somatematica.com.br

- **CÁLCULO DO M.D.C. PELO PROCESSO DAS DIVISÕES SUCESSIVAS**

Nesse processo efetuamos várias divisões até chegar a uma divisão exata. O divisor desta divisão é o m.d.c. Acompanhe o cálculo do m.d.c.(48,30).

Regra prática:

1º) dividimos o número maior pelo número menor;
48 / **30** = **1** (com resto **18**)

2º) dividimos o divisor 30, que é divisor da divisão anterior, por 18, que é o resto da divisão anterior, e assim sucessivamente;
30 / **18** = **1** (com resto **12**)

18 / **12** = **1** (com resto **6**)

12 / **6** = 2 (com resto zero - divisão exata)

3º) O **divisor da divisão exata** é 6. Então **m.d.c.(48,30) = 6**.

Exemplo

```
int mdc(int fat1,int fat2)
{
   if(fat2==0)  //   <==== Condição de parada
      return fat1;
   else
      return mdc(fat2,(fat1%fat2));    //   <=== Chamada Recursiva
}

int main(int argc, char *argv[])
{
  printf("\nMDC(16,20)=%d",mdc(16,20));

  printf("\n\n");
  system("PAUSE");
  return 0;
}
```

Exercício

- 1-) Crie um programa recursivo que calcule a potência (potencia>=0) de um número inteiro.
- Dica: $2^4 = 2 * 2^3$
  $2^3 = 2 * 2^2$
  $2^2 = 2 * 2^1$
  $2^1 = 2 * 2^0$

Exercícios

- Criar um função recursiva para calcular o fatorial de um número
  - Fonte: infoescola.com

O fatorial de um número **n** (n pertence ao conjunto dos números naturais) é sempre o produto de todos os seus antecessores, incluindo si próprio e excluindo o zero. A representação é feita pelo númeor fatorial seguido do sinal de exclamação, **n!** . Exemplo de número fatorial:

6! = 6 . 5 . 4 . 3 . 2 . 1 = 720

Importante: n >= 0 (n maior ou igual a zero) , ou seja, não existe fatorial para números negativos.

* O fatorial de 0 (**0!**) é 1, pois **o produto de número nenhum é 1**.

- Applet: http://ccism.pc.athabascau.ca/html/lo/repos/comp272/applets/factorial/index.html

Quando não usar recursividade

- Mão da massa:
  - Implementem o algoritmo para calcular a série de Fibonacci (1 1 2 3 5 8 13 ....) pelos métodos iterativos e recursivos.
  - Calcule o tempo de execução dos dois programas acima para as seguintes entradas: 10,20,30,50 e 100
  - O que você percebeu?
  - Lendo o capítulo 2.2 do livro texto (Zivian, Projeto de Algoritmos), responda quando não utilizar a recursividade?

Exercícios Recursão

- Dada a sequinte função recursiva:

```
int f(int n, int m) {
  if (n == 0) return m;
  else return f(n-1, 1-m);
}
```

Qual o resultado para f(3,7)?
(a) 0
(b) 3
(c) 7
(d) -7
(e) Nenhum dos anteriores

Resolução

(e)

f(3,7)
= f(2,-6)
= f(1, 7)
= f(0, -6)
= -6

Exercícios Recursão

Dada a seguinte função recursiva:

```
int g(int n) {
   if (n == 0) return 1
   else return g(n-1) + g(n-2);
}
```

Qual alternativa é correta?

(a) Não temos caso base

(b) Não é recursivo

(c) Essa função sempre termina para qualquer n

(d) Nenhuma das anteriores

Resolução

(d)

g(n) onde n é termina com 0

g(0) = 1

Nenhum outro valor termina.

g(-1) não termina,

g(1) logo, também não termina

Exercícios Recursão

```
int method( int n) {
   if (n < = 1) return 1;
   return method(n/2) + method(1);
}
```

A complexidade da função method é:

(a) O(log n)

(b) O(n)

(c) $O(n^2)$

(d) $O(2^n)$

(e) $O(n2^n)$

Resolução

(a)

O domínio é reduzido pela metade a cada chamada recursiva

Complexidade - Recorrências

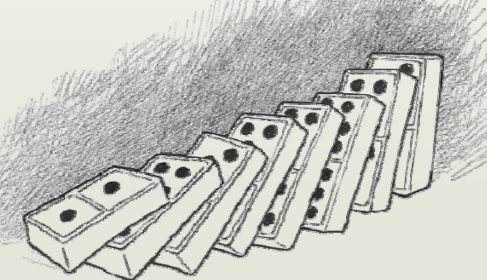
- Três métodos para resolvrmos recorrências
  - Substituição
    - Supomos um limite hipotético
    - Usamos a indução matemátic para provar que a suposição está correta
  - Árvore de recursão
    - Converte a recorrência em uma árvore
    - Cada nó representa o custo envolvido
  - Mestre
    - Fornece limites para a recorrência
    - Requer memorização de 3 casos

Substituição

- Pressupões a forma da solução
- Usa a indução matemática para encontrar constantes e mostrar que a solução funciona
- So pode ser usado quando é fácil pressupor a forma da resposta

Indução Matemática

- O *Método de Indução Matemática* é um método de demonstração elaborado com base no Princípio de Indução Finita, frequentemente utilizado para provar que certas propriedades são verdadeiras para todos os números naturais. (fonte: e-escola)
- Passos:
  - base da indução
  - hipótese da indução
  - passo da indução



Exercícios

- **Base da indução:** Para qualquer inteiro positivo
  - $n < 2^n$
  - **Provando:** para n=1 $1<2^1$ $1<2$ temos a base verdadeira

- **Hipótese da indução:** suponhamos que seja verdade para qualquer k inteiro positivo:
  - $k < 2^k$
- **Passo indução:** Para o sucessor de k (k+1), desejamos provar que:
  - $2k < 2.2^k = 2^{k+1}$

  - $(k+1) < 2^{k+1}$
  - $(k+1) < 2k$ e $2k<2^{k+1}$
  - $(k+1) < 2^{k+1}$

Complexidade

- Não é simples o calculo da complexidade em algoritmos recursivos
- Vamos tomar o exemplo abaixo:

```
void FacaAlgo(int &vetor, int esquerda,int direita)
 {
    int meio = (esquerda+ direita)/2;
    if (esquerda < direita) {
       FacaAlgo (vetor, esquerda,mid);
       FacaAlgo (vetor, meio +1, direita);
      Combinar(vetor, esquerda, meio , direita);
    }
 }
```

- **FacaAlgo**
  - **Função executada em T(n)**
  - **Vetor [left-right] = n**

- **Demos a sanguinte relação:**
  - T(n) = 2 T(n/2) + *O(n)* [O(n) é a combinação]
  - T(1) = O(1)

- relação de recorrência:
  - T(1) = 1
  - T(n) = 2.T(n/2) + n , para $n \geq 2$
- Resolvendo...
  - 1. Chute: T(n) = n + n.logn
  - 2. Prova:
    - 1. Caso base: 1 + 1.log 1 = 1
    - 2. H.I.: assumir que é válido para valores até n-1
    - 3. Provar T(n):
      - = 2.(n/2 + n/2.log n/2) + n
      - = n + n.(logn -1) + n
      - = n + n.logn
      - Logo, T(n) é O(n.logn)

- Resolvendo a relação de recorrência
- $T(n) = 2\ T(n/2) + n$
- $= 2\ [\mathbf{2\ T(n/4) + n/2}] + n$
- $= 4\ T(n/4) + 2n$
- $= 4\ [\mathbf{2\ T(n/8) + n/4}] + 2n$
- $= 8\ T(n/8) + 3n$
- = (*Qual a próxima sentença?*)
- $= 16\ T(n/16) + 4n$
- $= 2^k\ T(n/2^k) + k\ n$

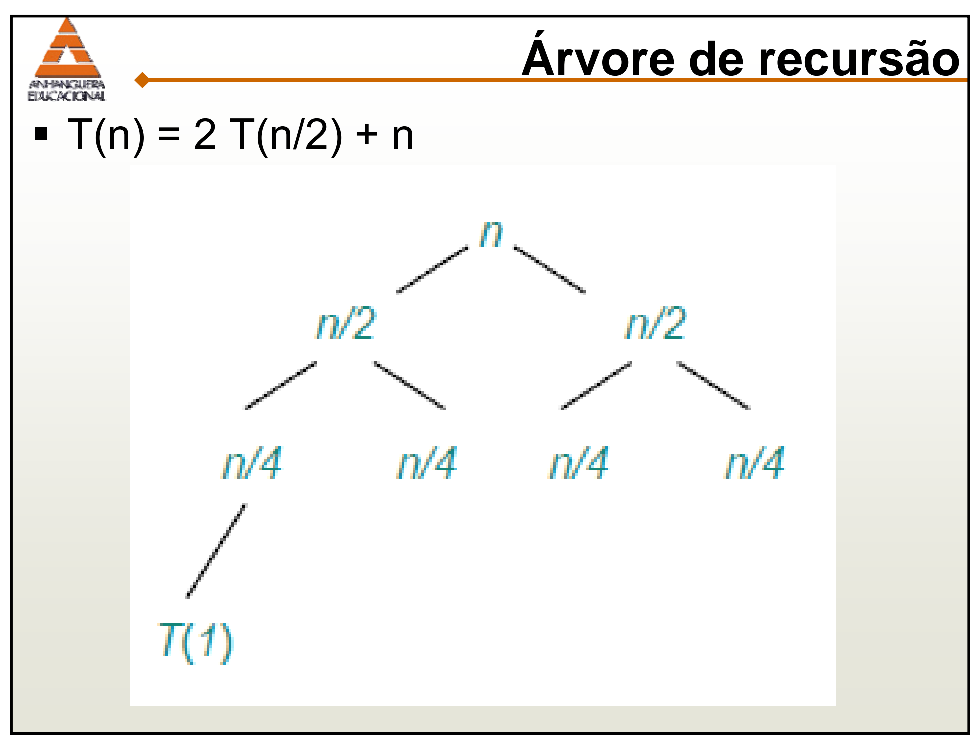
- Resolvendo a relação de recorrencia
- $T(n) = 2\ T(n/2) + n$
- $= 2\ [\mathbf{2\ T(n/4) + n/2}] + n$
- $= 4\ T(n/4) + 2n$
- $= 4\ [\mathbf{2\ T(n/8) + n/4}] + 2n$
- $= 8\ T(n/8) + 3n$
- = (*Qual a próxima sentença?*)

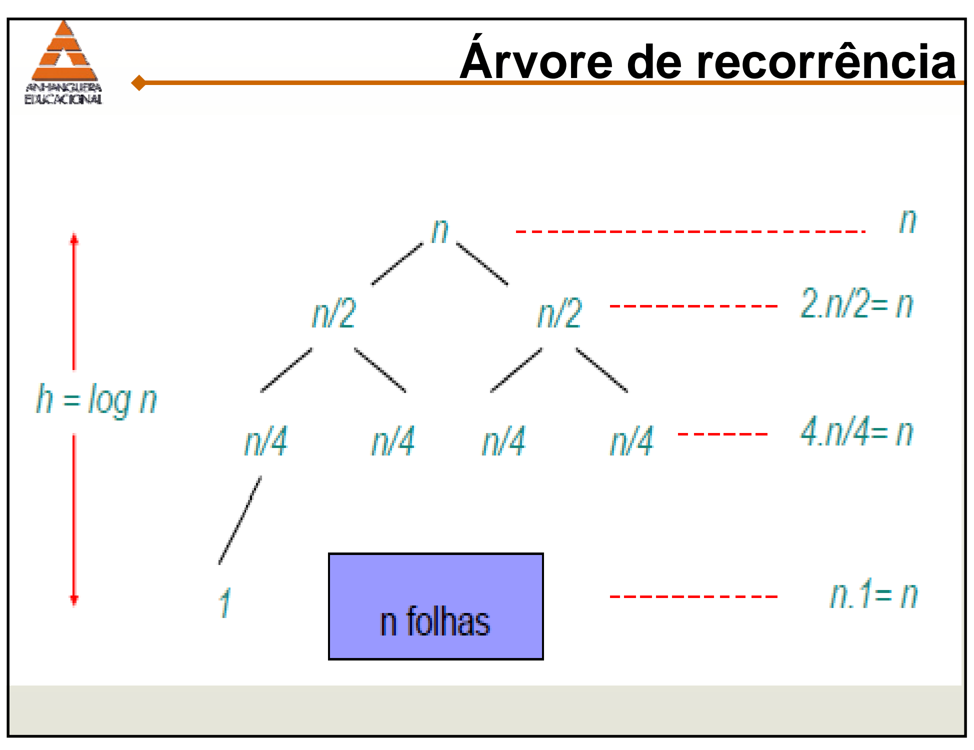
- Generalizando
  - $n/2^k = 1$ *OU* $n = 2^k$ OU $\log_2 n = k$
  - $k = \log_2 n$

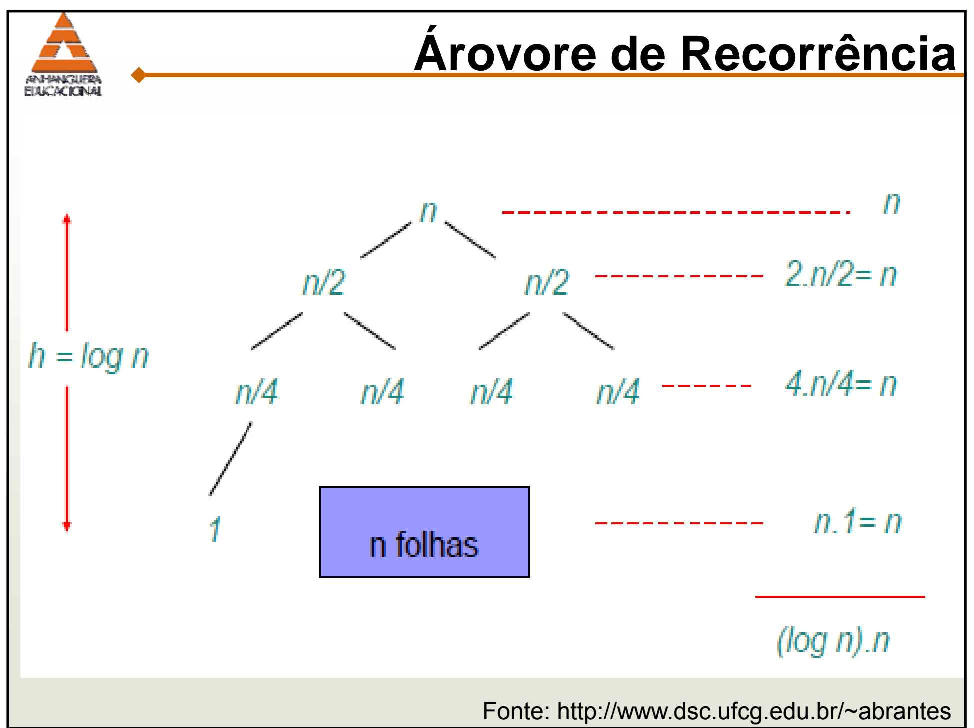
  - $= 2^k T(n/2^k) + k n$
  - $= 2^{\log_2 n} T(1) + (\log_2 n) n$
  - $= n + n \log_2 n$ [*lembrando T(1) = 1*]
  - $= O(n \log n)$

Árvore de Recorrência

- Desenhar a árvore
  - Cada nó representa o tamanho do problema
- Levar em conta
  - Altura da árvore
  - Número de passos em cada nível
- Solução
  - Soma dos passos de todos os níveis

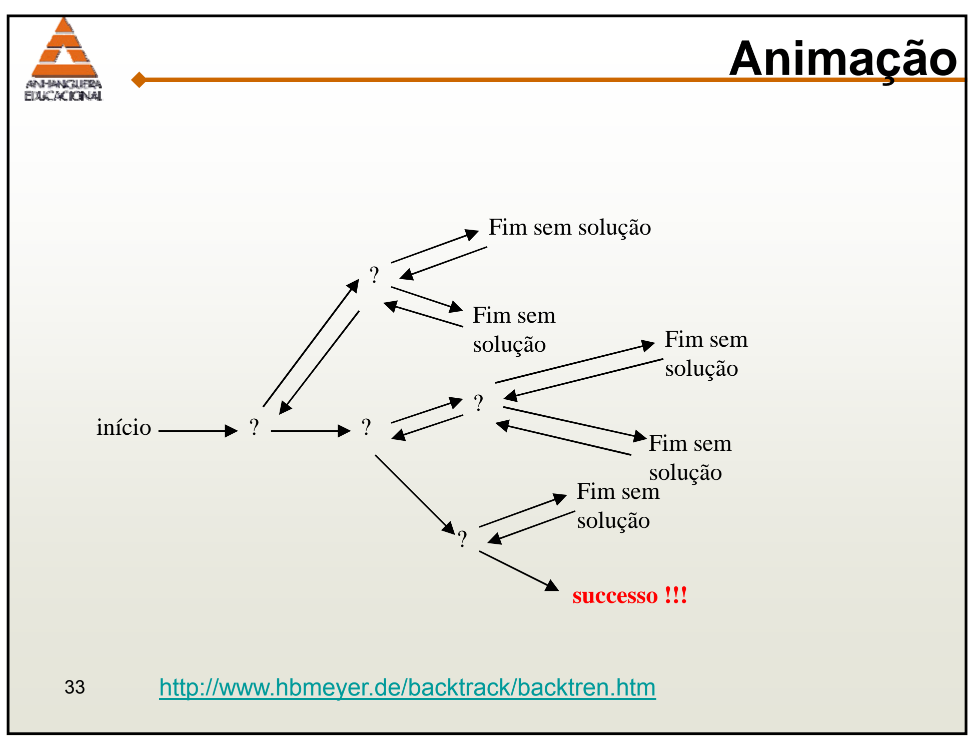

ANHANGUERA EDUCACIONAL
Árvore de recursão
T(n) = 2 T(n/2) + n
n
n/2
n/2
n/4
n/4
n/4
n/4
T(1)

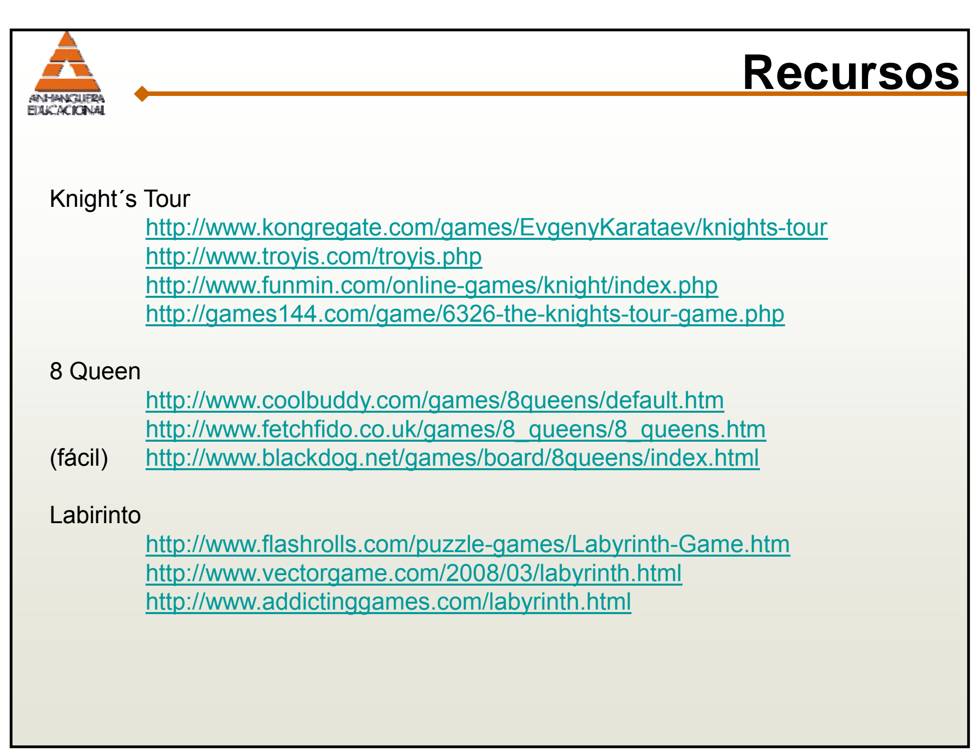

ANHANGUERA EDUCACIONAL
Árvore de recorrência
n
n
n/2
n/2
2.n/2= n
h = log n
n/4
n/4
n/4
n/4
4.n/4= n
1
n folhas
n.1= n



Tentativa e Erro

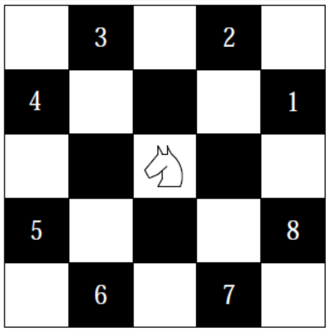
- É um caminho metódico de tentativa de decisões para verificar se o objetivo é alcançado
- Cenário
  - Devo tomar decisões entre várias escolhas possíveis
    - Não tenho informações para “escolher corretamente”
    - Cada decisão me leva a um novo conjunto de escolhas
    - Algumas seqüências de decisões podem resolver o problema
- Força - Bruta


Animação
início
?
?
?
Fim sem solução
Fim sem
solução
?
Fim sem
solução
Fim sem
solução
Fim sem
solução
?
successo !!!
33
http://www.hbmeyer.de/backtrack/backtren.htm

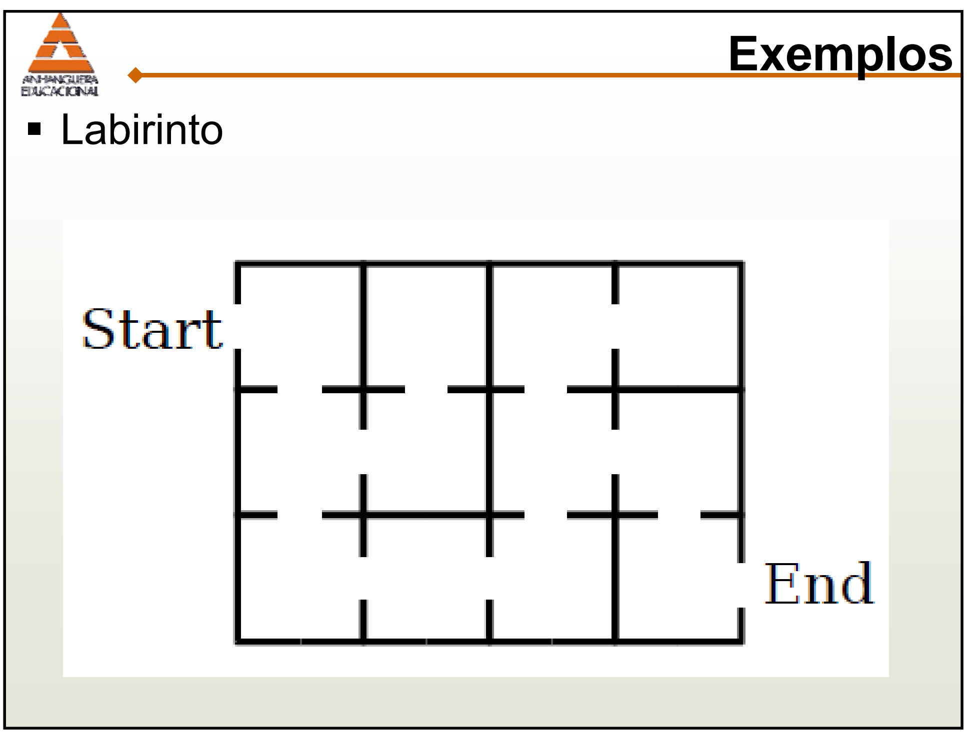

Recursos
Knight´s Tour
http://www.kongregate.com/games/EvgenyKarataev/knights-tour
http://www.troyis.com/troyis.php
http://www.funmin.com/online-games/knight/index.php
http://games144.com/game/6326-the-knights-tour-game.php
8 Queen
http://www.coolbuddy.com/games/8queens/default.htm
http://www.fetchfido.co.uk/games/8_queens/8_queens.htm
(fácil) http://www.blackdog.net/games/board/8queens/index.html
Labirinto
http://www.flashrolls.com/puzzle-games/Labyrinth-Game.htm
http://www.vectorgame.com/2008/03/labyrinth.html
http://www.addictinggames.com/labyrinth.html

Estretégia

- Tabuleiro com n n posições:
  - cavalo se movimenta segundo regras do xadrez.
- Problema: a partir de (x0; y0), encontrar, se existir, um passeio do cavalo que visita todos os pontos do tabuleiro uma única vez.
- Tenta um próximo movimento:
- (fonte: Ziviani, Nivio
- http://www.dcc.ufmg.br/algoritmos/)




```
procedure Tenta;
begin
inicializa selecao de movimentos;
repeat
  seleciona proximo candidato ao movimento;
  if aceitavel
  then begin
       registra movimento;
       if tabuleiro nao esta cheio
       then begin
            tenta novo movimento;
            if nao sucedido then apaga registro anterior;
            end;
       end;
until (mov. bem sucedido) ou (acabaram-se cand. movimento);
end;
```

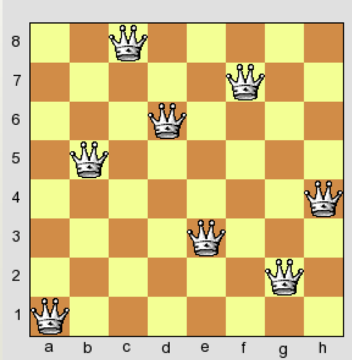


Estretégia

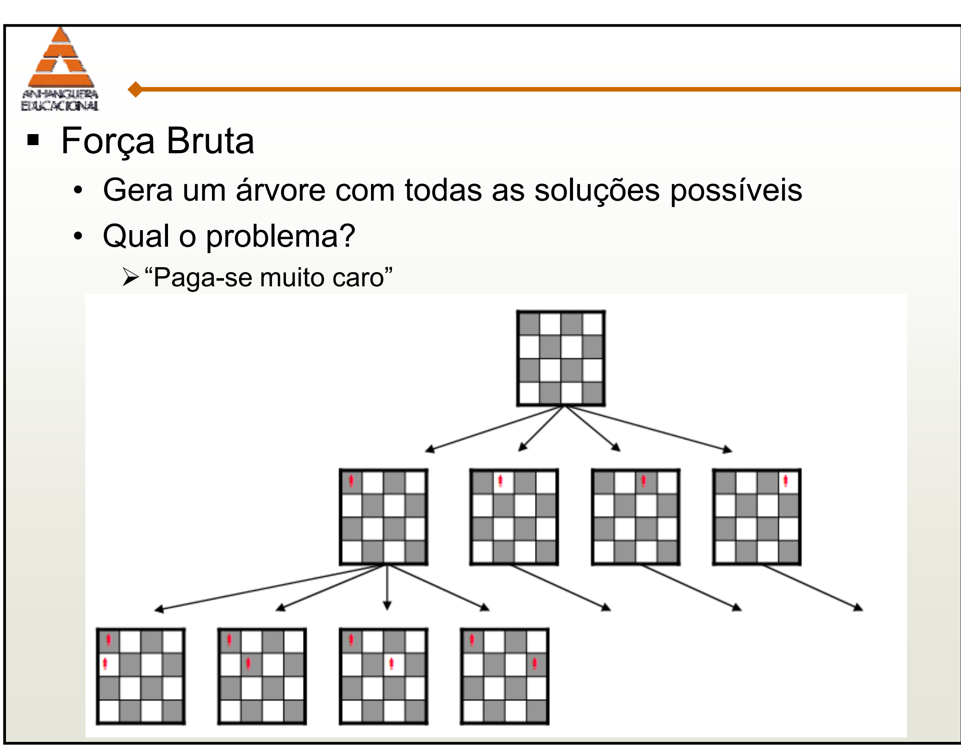
- Nome da Célula
  - Células $_{linha\ coluna}$
- Podem andar para cima,baixo,esquerda e direita
- Quando estou traçando o caminho, não volto para a mesma direção
- Refazer o caminho de qualquer celula $_{lin\ col}$ até o fim
  - Tento mover CIMA e tento resolver recursivamente a partir deste ponto
  - Tento BAIXO, ESQUERDA e DIREITA
- Para evitar loops infinitos, guardar os caminhos

N Rainhas

- Dado um tabuleiro NxN, colocar N rainhas no tabuleiro de forma que nenhuma ameace(*) a outra
  - Uma rainha esta ameaçada quando ela está na mesma linha, coluna ou diagonal

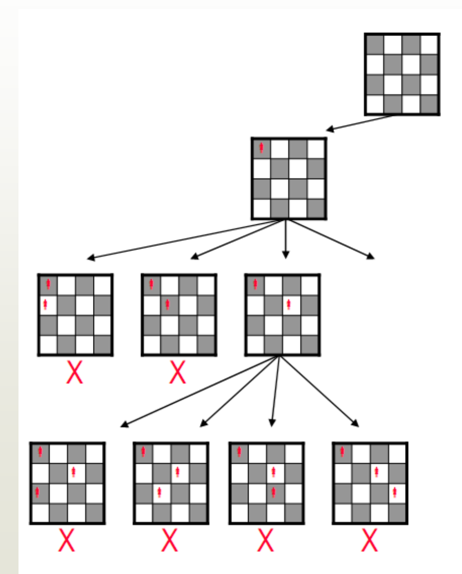


- Força Bruta
  - Gera um árvore com todas as soluções possíveis
  - Qual o problema?
    - “Paga-se muito caro”



- Tentativa e Erro (Backtracking)
  - Cria outra sub-árvore somente se não achar a solução

  Pode achar mais que uma solução



Exercícios

- Implemente uma solução para o sudoku utilizando a técnica de tentativa e erro.

Divisão e Conquista

- Divide o problema em partes menores,
- Encontrar a solução para as partes
- Combiná-las na solução geral

Exemplo

- Encontrar o maior e o menor elemento de um conjunto

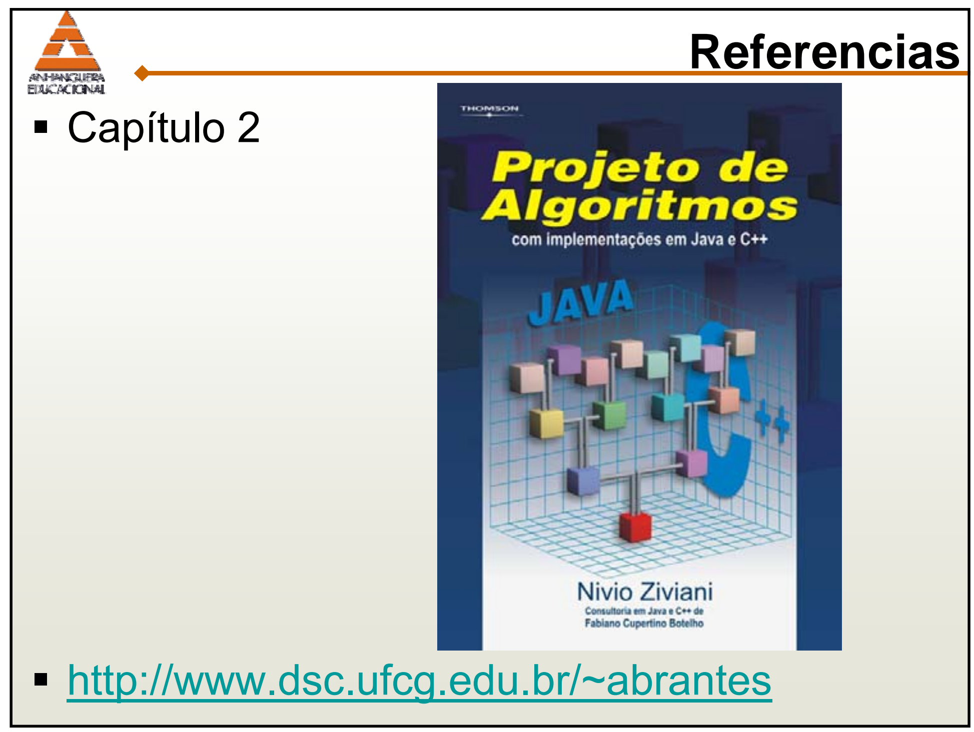
```
procedure MaxMin4(Linf,Lsup:integer; var Max,Min:integer)
var Max1, Max2, Min1, Min2, Meio: integer;
begin
  if Lsup – Linf <= 1
  then if A[Linf] < A[Lsup]
       then begin Max := A[Lsup]; Min := A[Linf]; end
       else begin Max := A[Linf]; Min := A[Lsup]; end
  else begin
       Meio := (Linf + Lsup) div 2;
       MaxMin4(Linf, Meio, Max1, Min1);
       MaxMin4(Meio+1, Lsup, Max2, Min2);
       if Max1 > Max2 then Max := Max1 else Max := Max2;
       if Min1 < Min2 then Min := Min1 else Min := Min2;
       end;
end;
```

Programação dinâmica

- Quando a soma dos tamanhos dos subproblemas é O(n) então é provável que o algoritmo recursivo tenha **complexidade polinomial.**
- Quando a divisão de um problema de tamanho n resulta em n subproblemas de tamanho n - 1 então é provável que o algoritmo recursivo tenha **complexidade exponencial.**
- Nesse caso, a técnica de programação dinâmica pode levar a um algoritmo mais eficiente.
- A programação dinâmica 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.
- (Ziviani) – Projeto de Algoritmos

Algoritmos Gulosos e de Aproximação

- Resolve problemas de otimização.
- Exemplo: algoritmo para encontrar o caminho mais curto entre dois vértices de um grafo.
- **– Escolhe a aresta que parece mais** promissora em qualquer instante;
- **– Independente do que possa acontecer** mais tarde, nunca reconsidera a decisão.
- Não necessita avaliar alternativas, ou usar procedimentos sofisticados para desfazer decisões tomadas previamente.
- Problema geral: dado um conjunto C,determine um subconjunto S C tal que:
- **– S satisfaz uma dada propriedade P, e**
- **– S é mínimo (ou máximo) em relação a** algum critério .
- O **algoritmo guloso para resolver o problema** geral consiste em um processo iterativo em que S é construído adicionando-se ao mesmo
- elementos de C um a um.


ANHANGUERA EDUCACIONAL
Referencias
Capítulo 2
THOMSON
Projeto de Algoritmos
com implementações em Java e C++
JAVA
C++
Nivio Ziviani
Consultoria em Java e C++ de
Fabiano Cupertino Botelho
http://www.dsc.ufcg.edu.br/~abrantes