
SCC0601 - Introdução à Ciência de Computação II

Recursão

Definição

- Uma função é dita *recursiva* quando é definida em seus **próprios termos**, direta ou indiretamente
 - Dicionário Michaelis: *ato ou efeito de recorrer*
 - Recorrer: *Correr de novo por; tornar a percorrer. Repassar na memória; evocar.*
- É uma função **como qualquer outra**

Recursividade

- Utiliza-se uma função que permite **chamar a si mesma** (direta ou indiretamente)
- Exemplo: soma dos primeiros N inteiros
 - $S(5) = 5 + 4 + 3 + 2 + 1$
 - $\quad = 5 + S(4)$
 - $S(4) = 4 + S(3)$
 - $S(3) = 3 + S(2)$
 - $S(2) = 2 + S(1)$
 - $S(1) = 1$ (**solução trivial**)

Recursividade

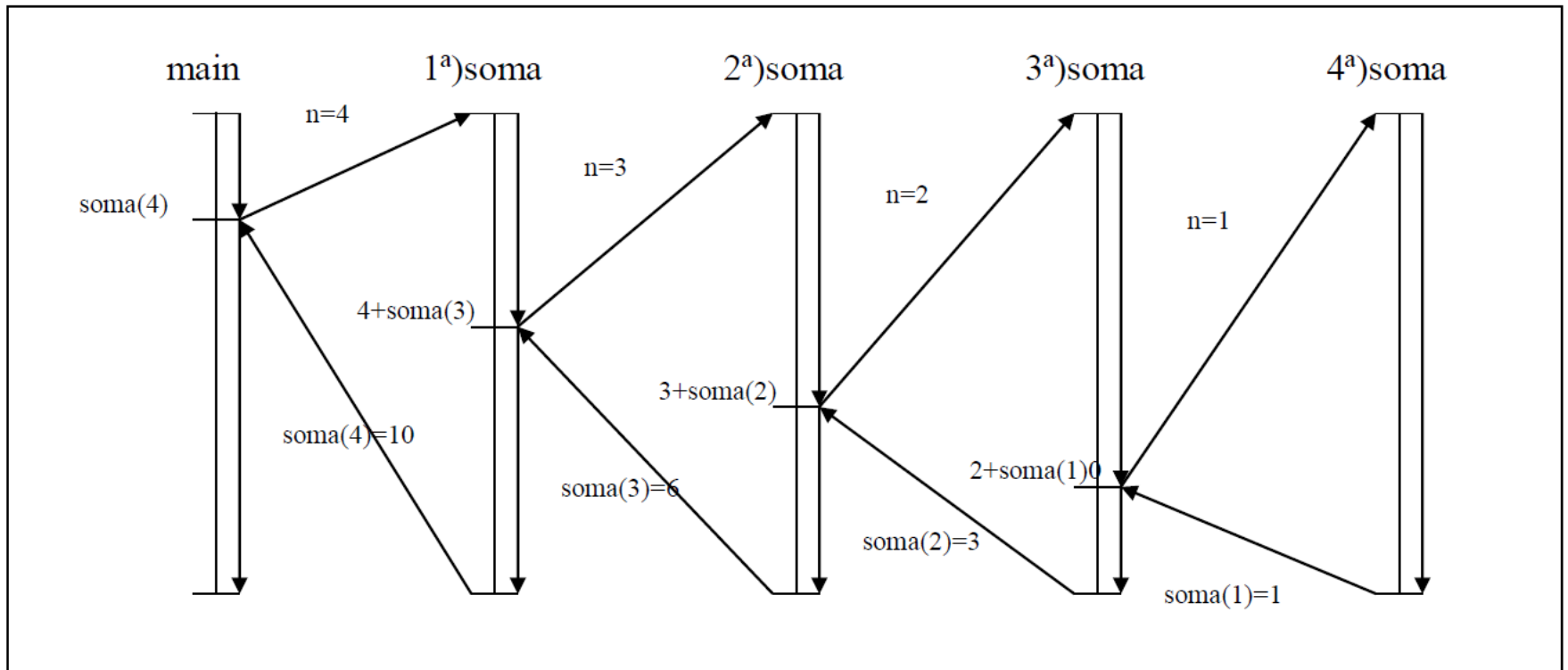
■ 2 passos:

- Definir uma **função recursiva**, que decresce até alcançar a solução mais simples (trivial)
 - Ex: $S(n) = n + S(n-1)$
- Definir a condição de **parada** (solução trivial)
 - Ex.: $S(1) = 1$

Recursividade

```
main( )
{
    int n;
    scanf("%d", &n);
    printf("%d", soma(n));
}
int soma(int n)
{
    if (n == 1) return (1);
    else return (n + soma(n - 1));
}
```

Recursividade



Exercício

- Implemente uma função recursiva para calcular o fatorial de um número inteiro positivo

Exercício

- Implemente uma função recursiva para calcular o fatorial de um número inteiro positivo

```
//versão recursiva
int fatorial(int n) {
    int fat;

    if (n==0) fat=1;
    else fat=n*fatorial(n-1);

    return(fat);
}
```


Exemplo

- Função que imprime os elementos de um vetor

```
void imprime(int v[], int tamanho) {  
    int i;  
    for (i=0;i<tamanho;i++)  
        printf("%d ",v[i]);  
}
```

Exercício

- Faça a versão recursiva da função

Exercício

- Faça a versão recursiva da função

```
void imprime(int v[], int tamanho, int indice_atual) {  
    if ( indice_atual < tamanho ) {  
        printf("%d ", v[indice_atual] );  
        imprime(v,tamanho,indice_atual+1);  
    }  
}
```

Efeitos da recursão

- A cada chamada
 - **Empilham-se** na memória os **dados locais** (variáveis e parâmetros) e o **endereço de retorno**
 - A função corrente só termina quando a função chamada terminar
 - **Executa-se a nova chamada** (que também pode ser recursiva)
 - Ao retornar, **desempilham-se** os dados da memória, restaurando o estado antes da chamada recursiva

Exercício para casa

- Simule a execução da função para um vetor de tamanho 3 e mostre a situação da memória a cada chamada recursiva

```
void imprime(int v[], int tamanho, int indice_atual) {  
    if (indice_atual < tamanho) {  
        printf("%d ", v[indice_atual]);  
        imprime(v, tamanho, indice_atual + 1);  
    }  
}
```

Alternativa: tem o mesmo efeito?

```
void imprime0(int v[], ...) {  
    printf("%d ",v[0]);  
    imprime1(v,...);  
}  
}
```

```
void imprime1(int v[], ...) {  
    printf("%d ",v[1]);  
    imprime2(v,...);  
}  
}
```

```
void imprime2(int v[], ...) {  
    printf("%d ",v[2]);  
    imprime3(v,...);  
}  
}  
...
```

Alternativa: tem o mesmo efeito?

```
void imprime0(int v[], ...) {  
    printf("%d ",v[0]);  
    imprime1(v,...);  
}  
}
```

```
void imprime1(int v[], ...) {  
    printf("%d ",v[1]);  
    imprime2(v,...);  
}  
}
```

```
void imprime2(int v[], ...) {  
    printf("%d ",v[2]);  
    imprime3(v,...);  
}  
}  
...
```

Mesmo resultado, com diferença de haver duplicação de código

Efeitos da recursão

- *Mesmo resultado, com diferença de haver duplicação de código*
 - O que isso quer dizer? **Funções recursivas são sempre melhores do que funções não recursivas?**

Efeitos da recursão

- *Mesmo resultado, com diferença de haver duplicação de código*
 - O que isso quer dizer? Funções recursivas são sempre melhores do que funções não recursivas?
 - **Depende do problema**, pois nem sempre a recursão é a melhor forma de resolver o problema, já que pode haver uma **versão simples e não recursiva** da função (que não duplica código e não consome mais memória)

Recursão

- **Quando usar:** quando o problema pode ser definido recursivamente de forma natural
- **Como usar**
 - 1º ponto: definir o problema de **forma recursiva**, ou seja, em termos dele mesmo
 - 2º ponto: definir a **condição de término** (ou *condição básica*)
 - 3º ponto: a cada chamada recursiva, deve-se tentar garantir que se está mais próximo de satisfazer a condição de término (**caso mais simples**)
 - Caso contrário, **qual o problema?**

Recursão

- Problema do fatorial

- 1º ponto: definir o problema de forma recursiva

- $n! = n * (n-1)!$

- 2º ponto: definir a condição de término

- $n=0$

- 3º ponto: a cada chamada recursiva, deve-se tentar garantir que se está mais próximo de satisfazer a condição de término

- A cada chamada, n é decrementado, ficando mais próximo da condição de término

Recursão vs. iteração

- Quem é **melhor**?

//versão recursiva

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

//versão iterativa

```
int fatorial(int n) {  
    int i, fat=1;  
  
    for (i=2;i<=n;i++)  
        fat=fat*i;  
  
    return(fat);  
}
```

Exercício

- Implemente uma função que verifique se uma dada string é um palíndromo
 - Implementação iterativa
 - Implementação recursiva

Exercício

- Implemente uma função recursiva para calcular o enésimo número de Fibonacci
 - 1º ponto: definir o problema de forma recursiva
 - 2º ponto: definir a condição de término
 - 3º ponto: a cada chamada recursiva, deve-se tentar garantir que se está mais próximo de satisfazer a condição de término

Exercício

- Implemente uma função recursiva para calcular o enésimo número de Fibonacci
 - 1º ponto: definir o problema de forma recursiva
 - $f(0)=0$, $f(1)=1$, $f(n)=f(n-1)+f(n-2)$ para $n \geq 2$
 - 2º ponto: definir a condição de término
 - $n=0$ e/ou $n=1$
 - 3º ponto: a cada chamada recursiva, deve-se tentar garantir que se está mais próximo de satisfazer a condição de término
 - n é decrementado em cada chamada

Recursão vs. iteração

- Quem é melhor? Simule a execução

//versão recursiva

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

//versão iterativa

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


Recursão vs. iteração

- Quem é melhor? Simule a execução

//versão recursiva

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

Certamente mais elegante, mas duplica muitos cálculos!

//versão iterativa

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

Recursão vs. iteração

- Quem é melhor?
 - Estimativa de tempo para Fibonacci (Brassard e Bradley, 1996)

<i>n</i>	10	20	30	50	100
Recursão	8 ms	1 s	2 min	21 dias	10 ⁹ anos
Iteração	1/6 ms	1/3 ms	1/2 ms	3/4 ms	1,5 ms

Fibonacci recursivo eficiente

- Crie um procedimento recursivo eficiente para calcular o n-ésimo termo da série de Fibonacci.
 - Evite chamadas recursivas desnecessárias

Fibonacci recursivo eficiente

- Crie um procedimento recursivo eficiente para calcular o n -ésimo termo da série de Fibonacci.
 - Evite chamadas recursivas desnecessárias
 - **Dica: armazene os valores da série já calculados em um vetor**

```
void fib( int v[], int n,
         int *current)
{
    //Caso simples
    if ( n == 1)
    {
        v[0] = 0;
        v[1] = 1;
        *current = 1;
        return ;
    }

    if ( n-1 > *current )
        fib( v,n-1,current );

    //Atualiza vetor
    v[n] = v[n-1] + v[n-2];
    *current = n;
}
```

```
int main()
{
    int v[50];
    int current = 0;

    //Chamando fib
    fib( v, 49, &current );

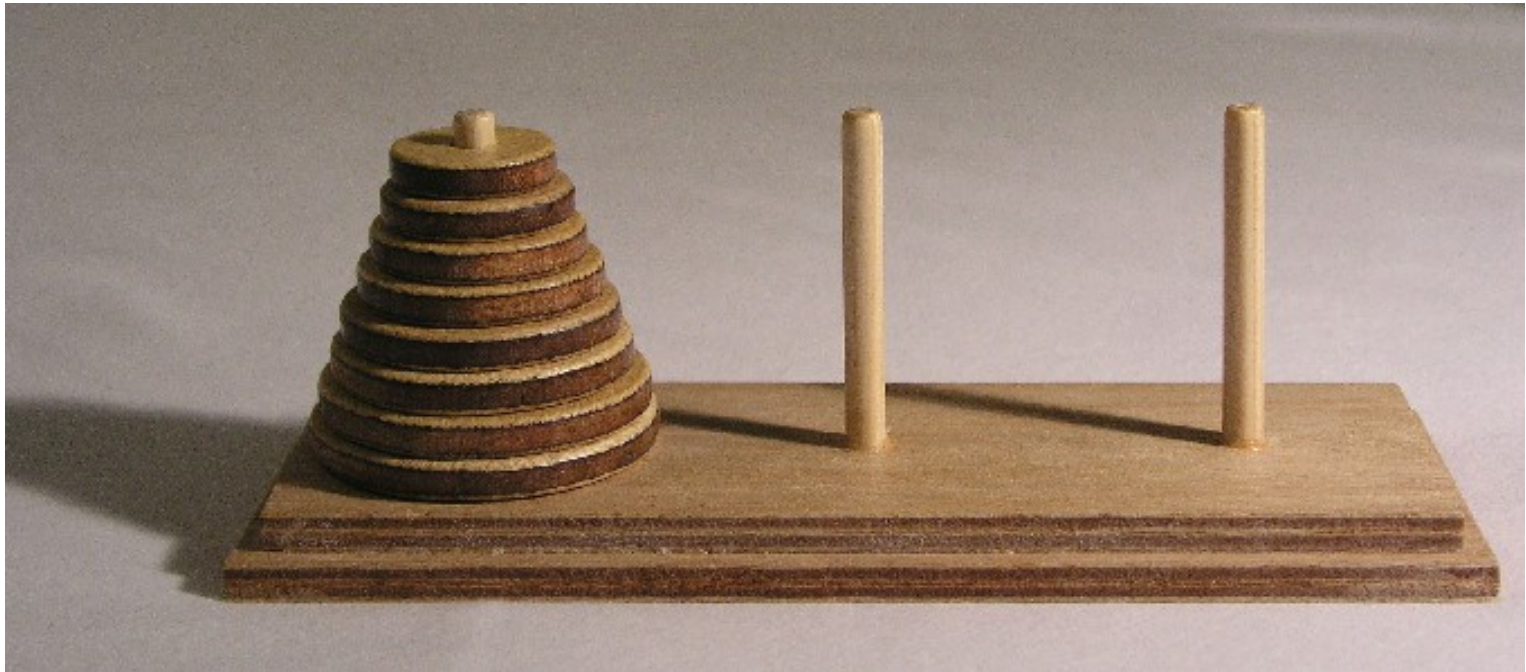
    int i;
    for ( i = 0; i < 50; i++)
        printf("%d\n", v[i]);

    return 0;
}
```

Recursão vs. iteração

- Programas recursivos que possuem chamadas ao **final do código** são ditos terem **recursividade de cauda**
 - São mais facilmente transformáveis em programas iterativos

Torres de Hanoi



Torres de Hanoi

- Jogo
 - Tradicionalmente com **3 hastes**: Origem, Destino, Temporária
 - Número qualquer de discos de tamanhos diferentes na haste Origem, dispostos em ordem de tamanho: os maiores embaixo
 - Objetivo: usando a haste Temporária, movimentar um a um os discos da haste Origem para a Destino, sempre respeitando a ordem de tamanho
 - **Um disco maior não pode ficar sobre um menor!**

Torres de Hanoi

- Implemente uma função recursiva que resolva o problema das Torres de Hanoi

Torres de Hanoi

- Implemente uma função recursiva que resolva o problema das Torres de Hanoi
 - Mover $n-1$ discos da haste Origem para a haste Temporária
 - Mover o disco n da haste Origem para a haste Destino
 - Recomeçar, movendo os $n-1$ discos da haste Temporária para a haste Destino

Torres de Hanoi

```
#include <stdio.h>

void mover(int, char, char, char);

int main(void) {
    mover(3,'O','T','D');
    system("pause");
    return 0;
}

void mover(int n, char Orig, char Temp, char Dest) {
    if (n==1) printf("Mova o disco 1 da haste %c para a haste %c\n",Orig,Dest);
    else {
        mover(n-1,Orig,Dest,Temp);
        printf("Mova o disco %d da haste %c para a haste %c\n",n,Orig,Dest);
        mover(n-1,Temp,Orig,Dest);
    }
}
```

Torres de Hanoi

- Exercício para casa

- Execute na mão o programa anterior
- Tente fazer a versão não recursiva do programa

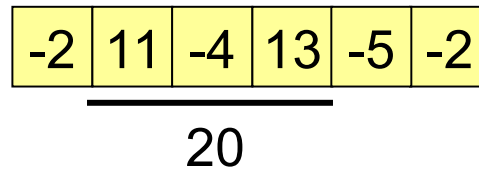
Multiplicação de matrizes

- Exercício para casa

- Tente fazer uma função recursiva que calcule a multiplicação de duas matrizes inteiras quadradas.

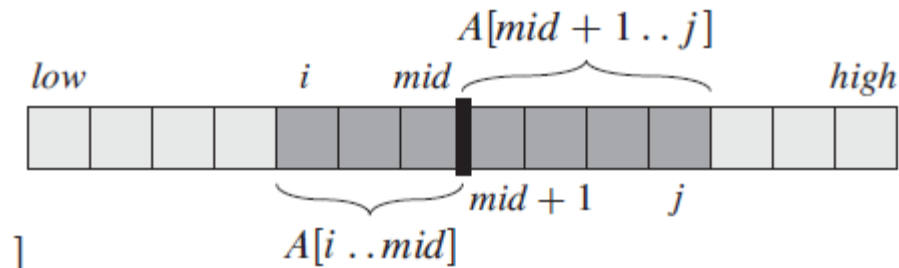
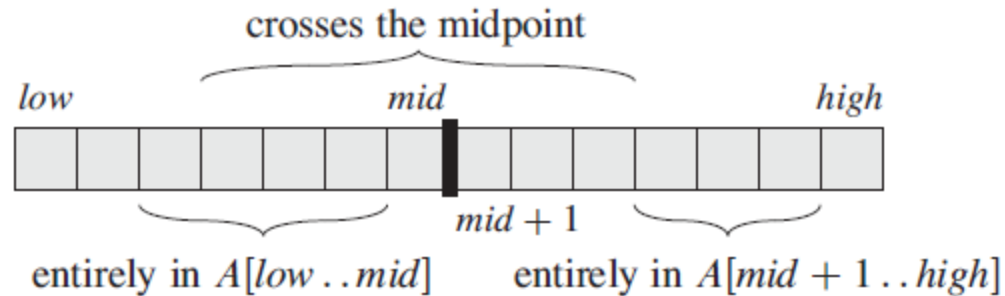
Exercício

- Problema da maior soma de subsequência em um vetor



- Implemente um programa iterativo para resolver o problema
- Implemente um programa recursivo

Duas possibilidades na divisão



Vetores cruzando o ponto central

FIND-MAX-CROSSING-SUBARRAY($A, low, mid, high$)

```
1  left-sum =  $-\infty$ 
2  sum = 0
3  for  $i = mid$  downto  $low$ 
4      sum = sum +  $A[i]$ 
5      if  $sum > left-sum$ 
6          left-sum = sum
7          max-left =  $i$ 
8  right-sum =  $-\infty$ 
9  sum = 0
10 for  $j = mid + 1$  to  $high$ 
11     sum = sum +  $A[j]$ 
12     if  $sum > right-sum$ 
13         right-sum = sum
14         max-right =  $j$ 
15 return (max-left, max-right, left-sum + right-sum)
```


Solução final

FIND-MAXIMUM-SUBARRAY ($A, low, high$)

```
1  if  $high == low$ 
2      return ( $low, high, A[low]$ )           // base case: only one element
3  else  $mid = \lfloor (low + high) / 2 \rfloor$ 
4      ( $left-low, left-high, left-sum$ ) =
           FIND-MAXIMUM-SUBARRAY( $A, low, mid$ )
5      ( $right-low, right-high, right-sum$ ) =
           FIND-MAXIMUM-SUBARRAY( $A, mid + 1, high$ )
6      ( $cross-low, cross-high, cross-sum$ ) =
           FIND-MAX-CROSSING-SUBARRAY( $A, low, mid, high$ )
7      if  $left-sum \geq right-sum$  and  $left-sum \geq cross-sum$ 
8          return ( $left-low, left-high, left-sum$ )
9      elseif  $right-sum \geq left-sum$  and  $right-sum \geq cross-sum$ 
10         return ( $right-low, right-high, right-sum$ )
11     else return ( $cross-low, cross-high, cross-sum$ )
```

Exercício

- Escreva uma função que retorne o segundo maior valor de uma função. Utilize recursão nesta função dividindo o vetor analisado ao meio à cada chamada.

Exercício

- Escreva imprima todas as $n!$ permutações da string de entrada. Considere que a string de entrada não possui caracteres repetidos e que ela está armazenada na primeira linha de M

```
void permute ( char *a, int ini, int length )
```

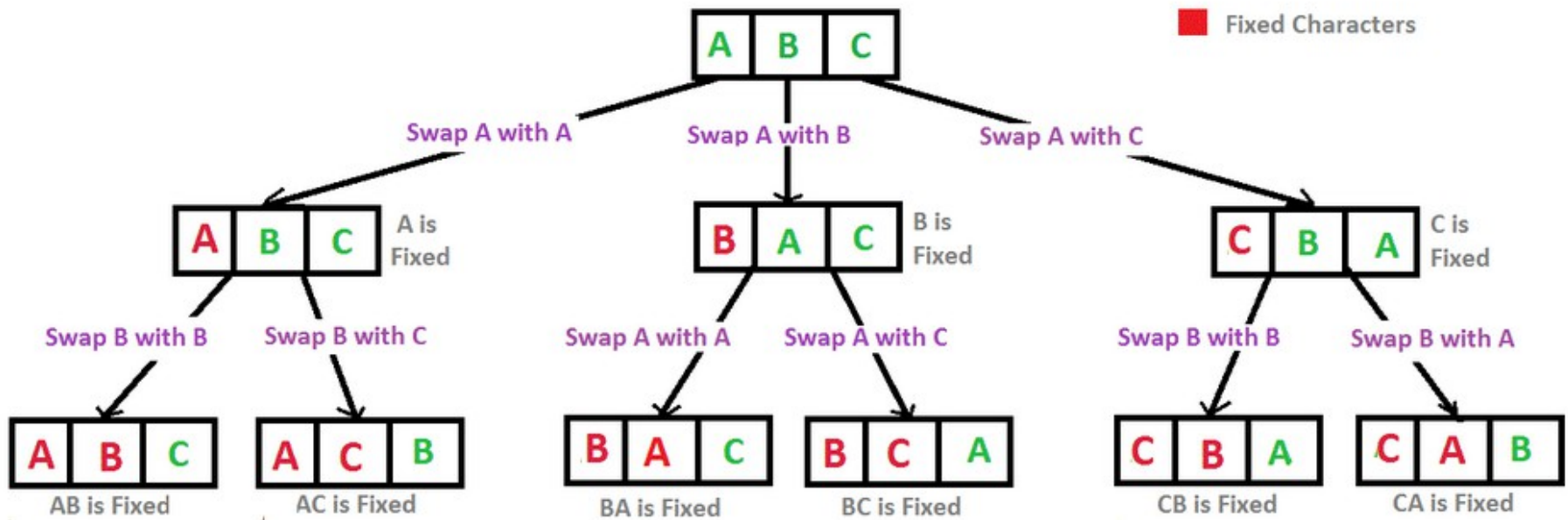
Exercício

- void permute (char *a, int ini, int length)
- Como definir de forma recursiva?

Exercício

- void permute (char *a, int ini, int length)
- Como definir de forma recursiva
 - Trocar o primeiro por cada um dos outros restantes da substring
 - Fazer permutação da substring

Árvore de recursão



```
void permute(char *a, int i, int n)
{
    int j;

    if ( i == n - 1) printf("%s\n", a);
    else
    {
        for (j = i; j < n; j++)
        {
            swap((a+i), (a+j));
            permute(a, i+1, n);
            swap((a+i), (a+j));
        }
    }
}
```

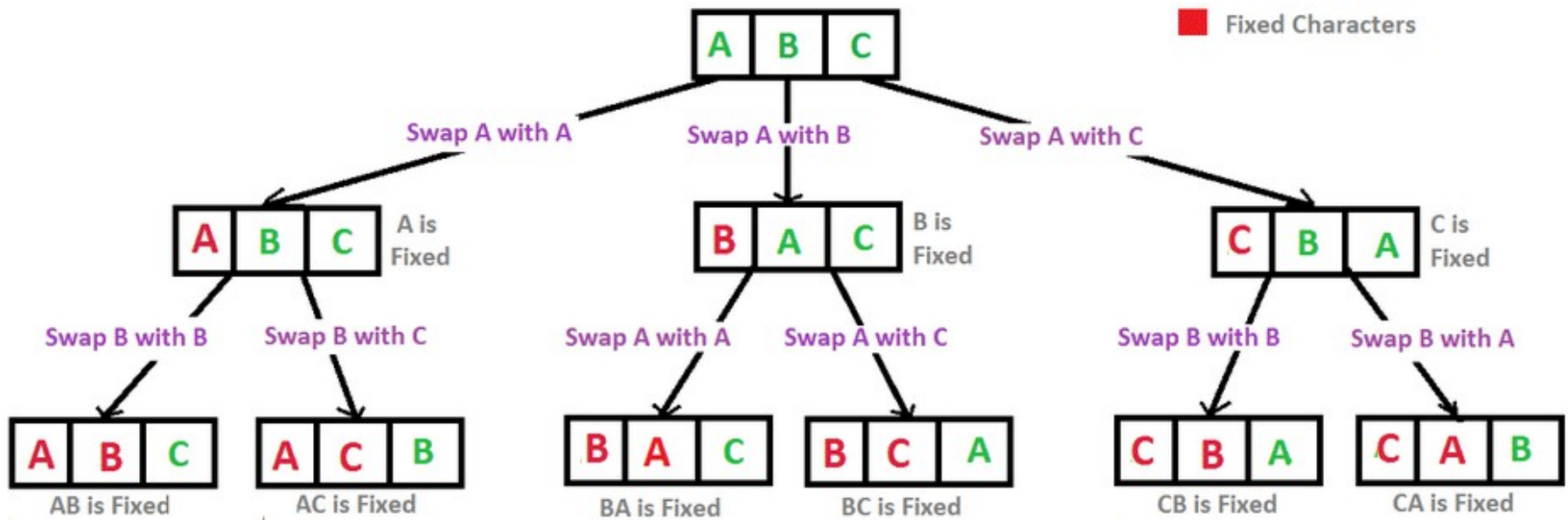
```
void swap(char *a, char *b)
{ char t = *a; *a = *b; *b = t; }

int main()
{

    char s[100];
    strcpy(s, "ABC");
    permute(s, 0, strlen(s));

    return 0;
}
```


Árvore de recursão

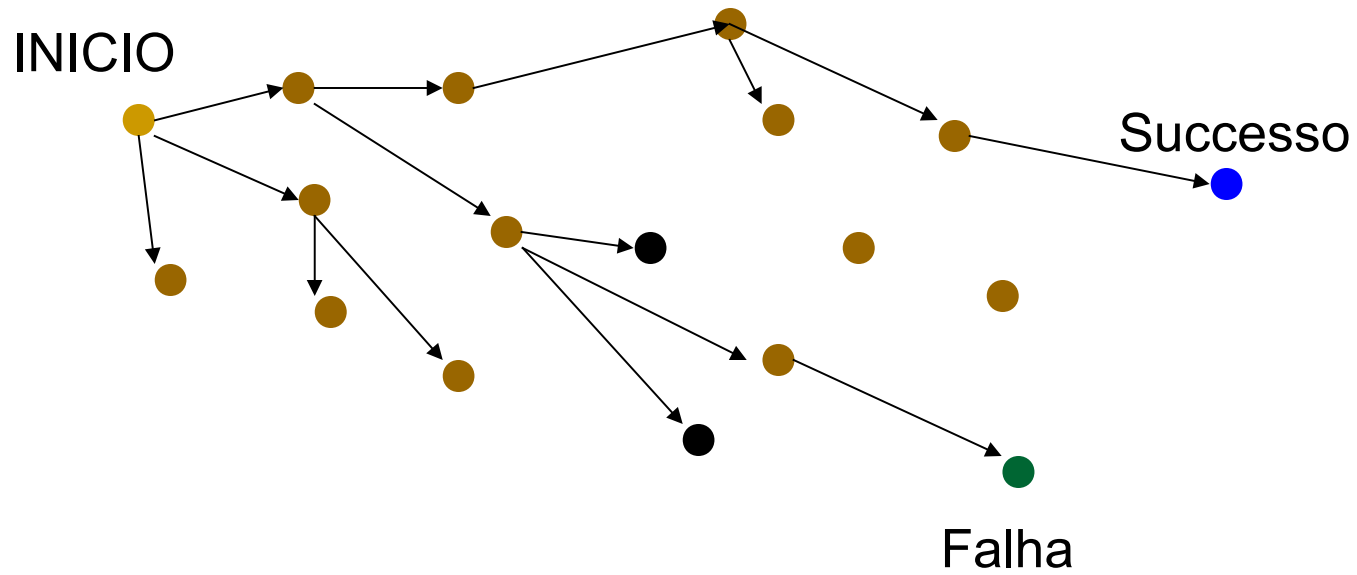


Backtracking

Backtracking

- Quando utilizar?
 - Problemas que **não seguem uma regra fixa** de cálculo, mas que são resolvidos por **tentativa e erro**
 - O problema pode ser decomposto em partes menores, através da **exploração de pequenas partes**. Pode ser visto como um algoritmo de busca em um dado espaço.
 - Se o próximo passo **não deu certo, volto** ao passo anterior e busca nova solução

Backtracking



Sudoku

- Matriz 9x9 com alguns números preenchidos
- Todos os números estão entre 1 e 9
- Objetivo: Cada linha, cada coluna, e cada mini matriz precisa conter os números entre 1 e 9 sem repetições

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Resolução via força bruta

- Solução **simples** e geral
- Tente **todas as combinações** até encontrar uma que funcione
- Esta abordagem não é “**esperta**”

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Resolução via força bruta

- Se não existem células vazias, resolvido
- Percorrer células da esquerda para direita, de cima para baixo
- Quando uma célula vazia é encontrada, preencher de 1 até 9
- Quando um dígito é colocado, verificar legalidade

5	3	1		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Resolução via força bruta

- Se não existem células vazias, resolvido
- Percorrer células da esquerda para direita, de cima para baixo
- Quando uma célula vazia é encontrada, preencher de 1 até 9
- Quando um dígito é colocado, verificar legalidade

5	3	1		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

ESTA NOVA INSTÂNCIA DO PROBLEMA É SIMILAR AO PROBLEMA ORIGINAL ?

5	3	1		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7	4			
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7	4	8		
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7	4	8	9	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

DEAD END

Sudoku - A Dead End

- Com a configuração atual, nenhum dos dígitos funciona para fechar a última linha

5	3	1	2	7	4	8	9	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Back tracking

- Quando a busca alcança um **ponto sem solução**, ela **volta** à solução anterior e muda para o próximo dígito
 - No exemplo, voltaríamos na célula anterior e colocaríamos um 9 e tentamos novamente

Back tracking

5	3	1	2	7	4	8	9	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7	4	9		
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Força bruta e backtracking

- Algoritmos de força bruta são **lentos**
- Não empregam muita lógica
 - Não enxergam muito à frente
- Mas ...
 - São **fáceis de implementar** como primeira solução
- Backtracking é um tipo de solução via força bruta

Características importantes

- Após tentar colocar um dígito na célula, queremos resolver o **mesmo subproblema**
- Após colocar um dígito na célula, é necessário lembrar o **próximo número a ser tentado** caso esta tentativa não funcione
- É necessário saber **se a solução funcionou ou não**. Se funcionou, não é necessário testar o próximo número
- Se todos os números são testados e nenhum deles funcionou, reportar

Pseudocódigo para backtracking

IF solução, RETURN sucesso

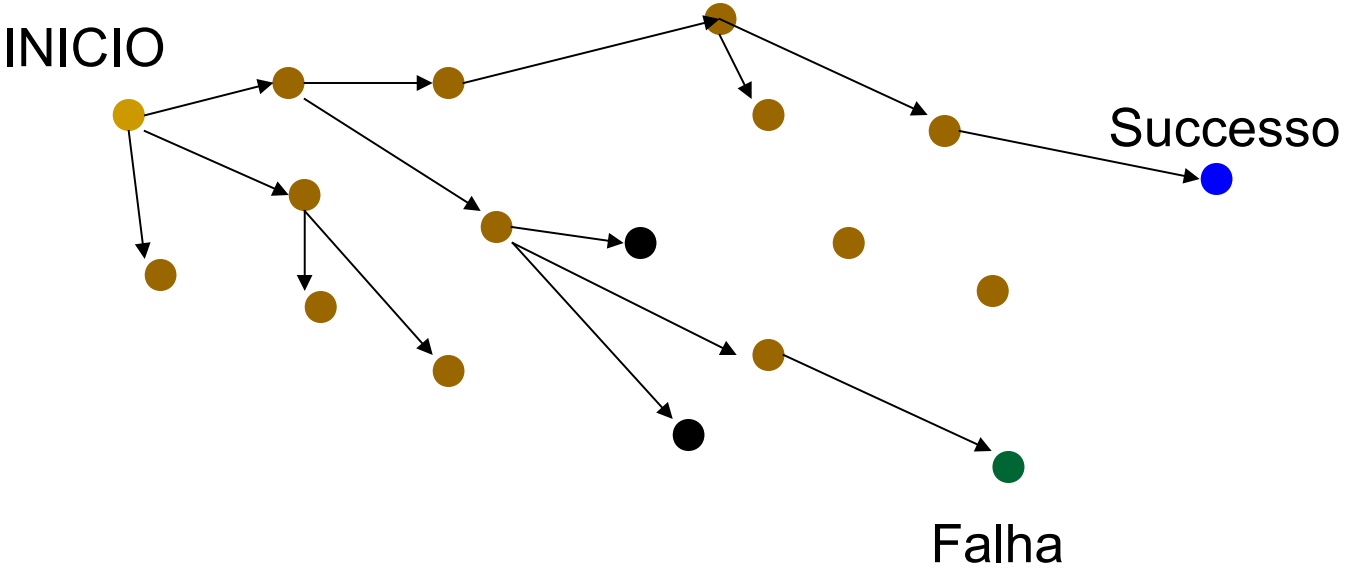
For (cada possível escolha do estado atual)

- faça a escolha e use o valor escolhido
- use recursão para resolver o problema para o novo estado

IF (chamada recursiva com sucesso)

RETURN sucesso

RETURN falha



Implementação com pilha

- Como implementar a solução do backtracking recursivo com pilha?

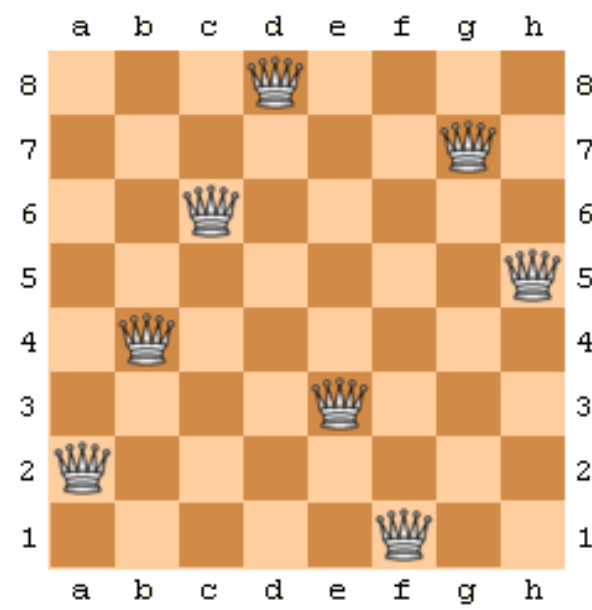
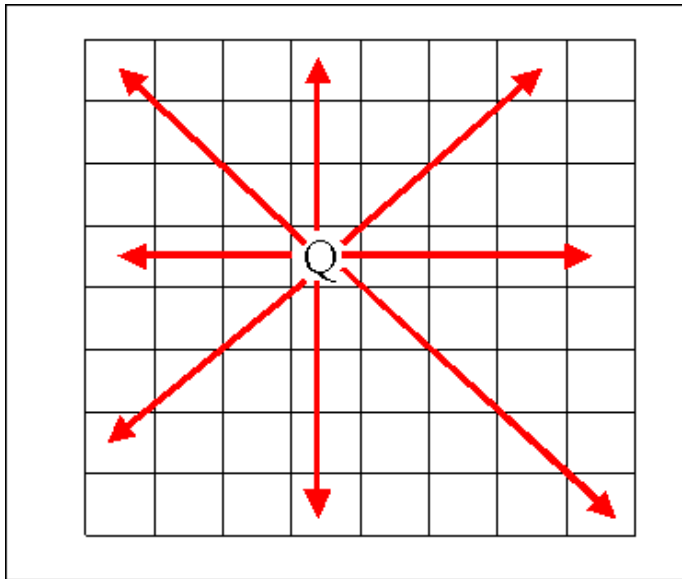
Implementação com pilha

```
boolean solve(Node n) {
    Coloque n na pilha;
    while (pilha não esta vazia)
    {
        if ( nó no topo é folha ) {
            if ( nó no topo é solucao), return true
            else pop()
        }
        else {
            if ( nó no topo tem filhos não verificados )
                coloque o próximo filho não verificado na pilha
            else pop()
        }
    }
    return false
}
```

O problema das 8 rainhas

Problema das 8 rainhas

- Um problema clássico de xadrez
 - Colocar 8 rainhas no tabuleiro de forma que nenhuma delas possa atacar as outras



O problema das N rainhas

- Colocar N rainhas em um tabuleiro N x N de forma que nenhuma delas possa atacar a outra
- Número de disposições possíveis
- Em 8 x 8 → Combinação de 64 tomados 8 a 8
= 4,426,165,368

- Se o número de rainhas é fixo e sabendo-se que não pode existir mais de uma rainha por coluna é possível iterar da seguinte forma

```
for(int c0 = 0; c0 < 8; c0++){
    board[c0][0] = 'q';
    for(int c1 = 0; c1 < 8; c1++){
        board[c1][1] = 'q';
        for(int c2 = 0; c2 < 8; c2++){
            board[c2][2] = 'q';
            // a little later
            for(int c7 = 0; c7 < 8; c7++){
                board[c7][7] = 'q';
                if( queensAreSafe(board) )
                    printSolution(board);
                board[c7][7] = ' '; //pick up
queen
            }
            board[c6][6] = ' '; // pick up queen
```

Problema

- É possível implementar a solução anterior para um valor variável N ?
 -

Problema

- É possível implementar a solução anterior para um valor variável N ?
 - Não é possível implementar uma quantidade genérica de loops
- Solução: implementar o problema recursivamente

Esquema geral da construção

```
PROCEDURE Try(i: INTEGER);
BEGIN
  initialize selection of positions for i-th queen:
  REPEAT make next selection;
    IF safe THEN SetQueen;
      IF i < 8 THEN Try(i+1);
        IF not successful THEN RemoveQueen END
      END
    END
  UNTIL successful OR no more positions
END Try
```

O problema das 8 rainhas

- i representa o índice da coluna e o processo de seleção para posições varia entre as 8 possíveis valores para linha.
- Forma de representação matricial?
 - Mais natural, no entanto esta representação leva a operações inconvenientes
 - O que é relevante para o problema?
 - Se aquela linha já foi ocupada
 - Se aquela diagonal já foi ocupada

Representação

```
VAR x: ARRAY 8 OF INTEGER;  
    a: ARRAY 8 OF BOOLEAN;  
    b, c: ARRAY 15 OF BOOLEAN
```

where

x_i denotes the position of the queen in the i th column;
 a_j means "no queen lies in the j th row";
 b_k means "no queen occupies the k th /-diagonal;
 c_k means "no queen sits on the k th \-diagonal.

- Para facilitar a resolução, note que
 - Diagonal / : elementos tem a soma constante
 - Diagonal \ : a diferença ($i - j$) é constante
- *SetQueen*: colocar a rainha na posição
 - $x[i] = j$
 - $a[j] = \text{FALSE}$ e $b[i+j] = \text{FALSE}$ e $c[i-j+7] = \text{FALSE}$
- *RemoveQueen*: remove a rainha da posição
 - $a[j] = \text{TRUE}$ e $b[i+j] = \text{TRUE}$ e $c[i-j+7] = \text{TRUE}$
- Safe: $a[j] \ \&\& \ b[i+j] \ \&\& \ c[i-j+7]$

```
PROCEDURE Try(i: INTEGER; VAR q: BOOLEAN);
  VAR j: INTEGER;
BEGIN j := 0;
  REPEAT q := FALSE;
    IF a[j] & b[i+j] & c[i-j+7] THEN
      x[i] := j;
      a[j] := FALSE; b[i+j] := FALSE; c[i-j+7] := FALSE;
      IF i < 7 THEN
        Try(i+1, q);
        IF ~q THEN
          a[j] := TRUE; b[i+j] := TRUE; c[i-j+7] := TRUE
        END
      ELSE q := TRUE
      END
    END ;
    INC(j)
  UNTIL q OR (j = 8)
END Try;
```

Resolução de labirintos

Exemplo: mazes

INÍCIO

X									
									X

SAÍDA

Exemplo: mazes

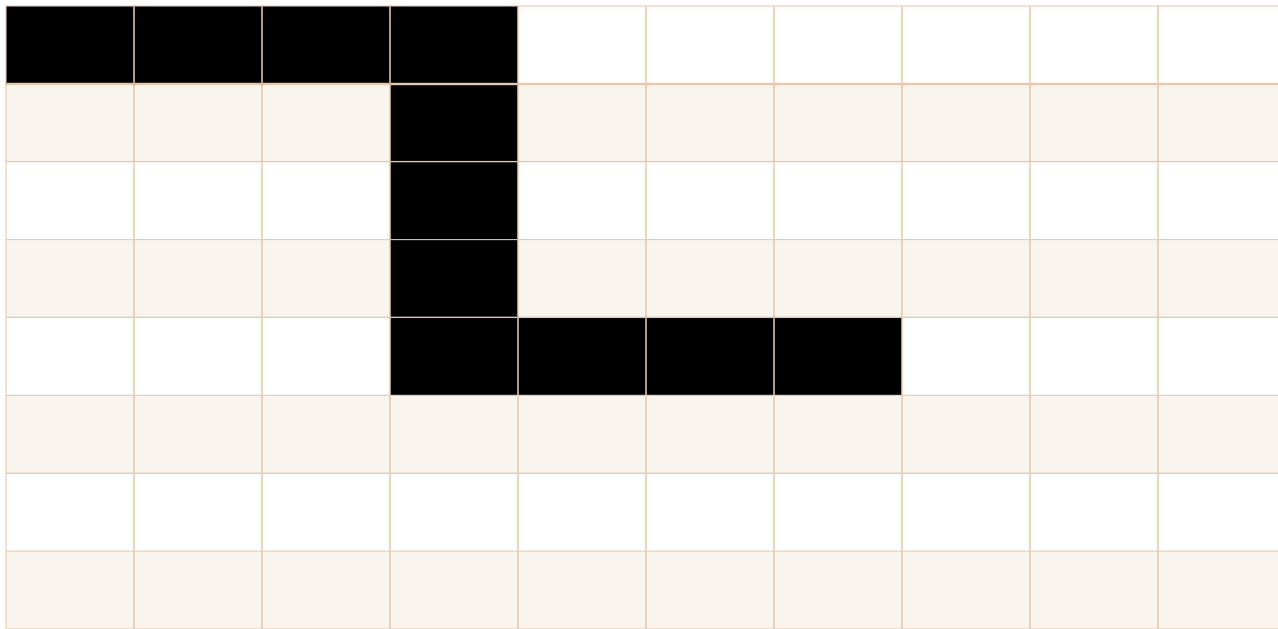
INÍCIO

█	█	█	█						
			█						
			█						
			█						
			█						

SAÍDA

Exemplo: mazes

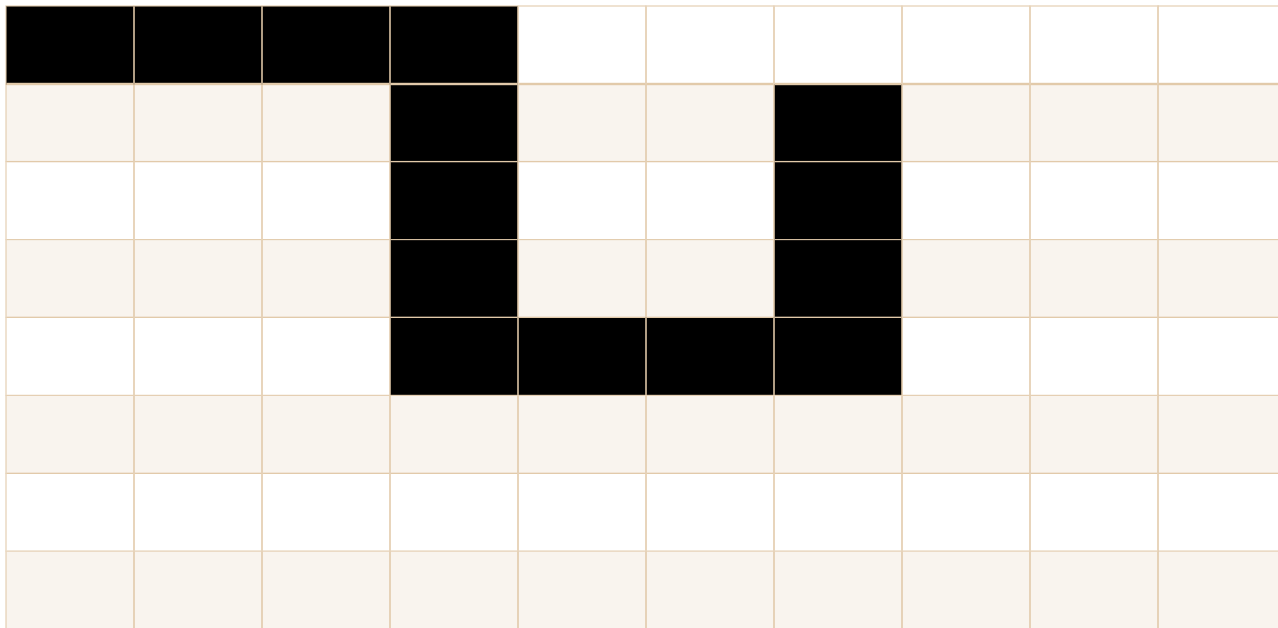
INÍCIO



SAÍDA

Exemplo: mazes

INÍCIO

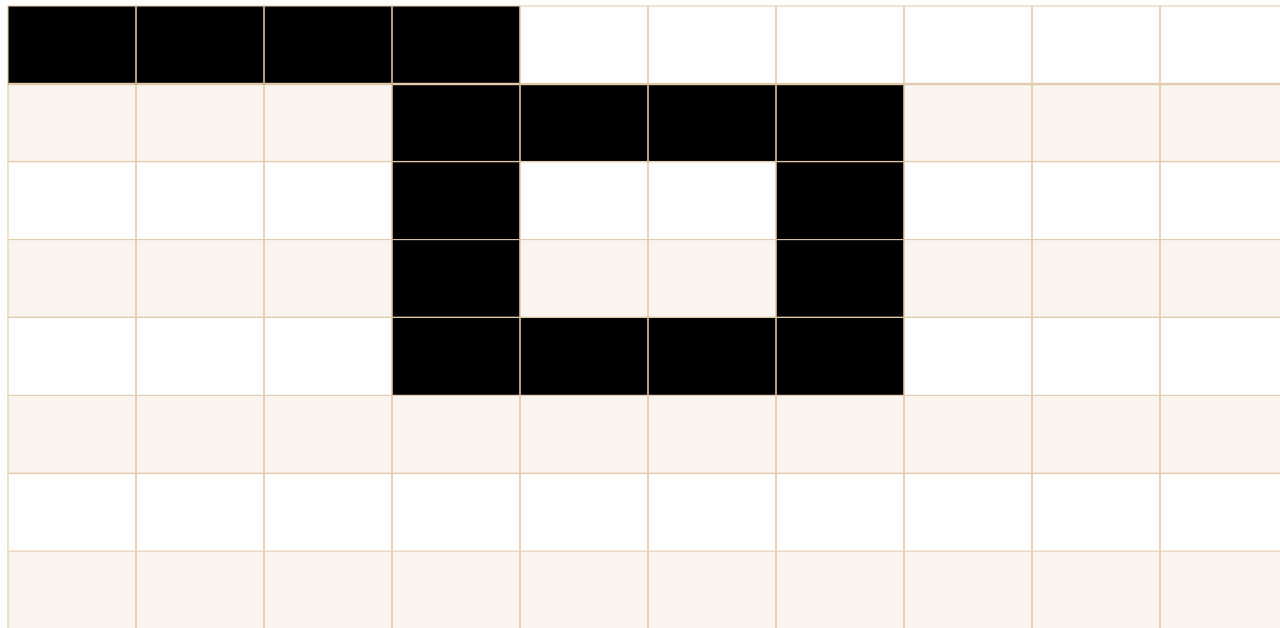


SAÍDA

Exemplo: mazes

CAMINHO ERRADO !

INÍCIO

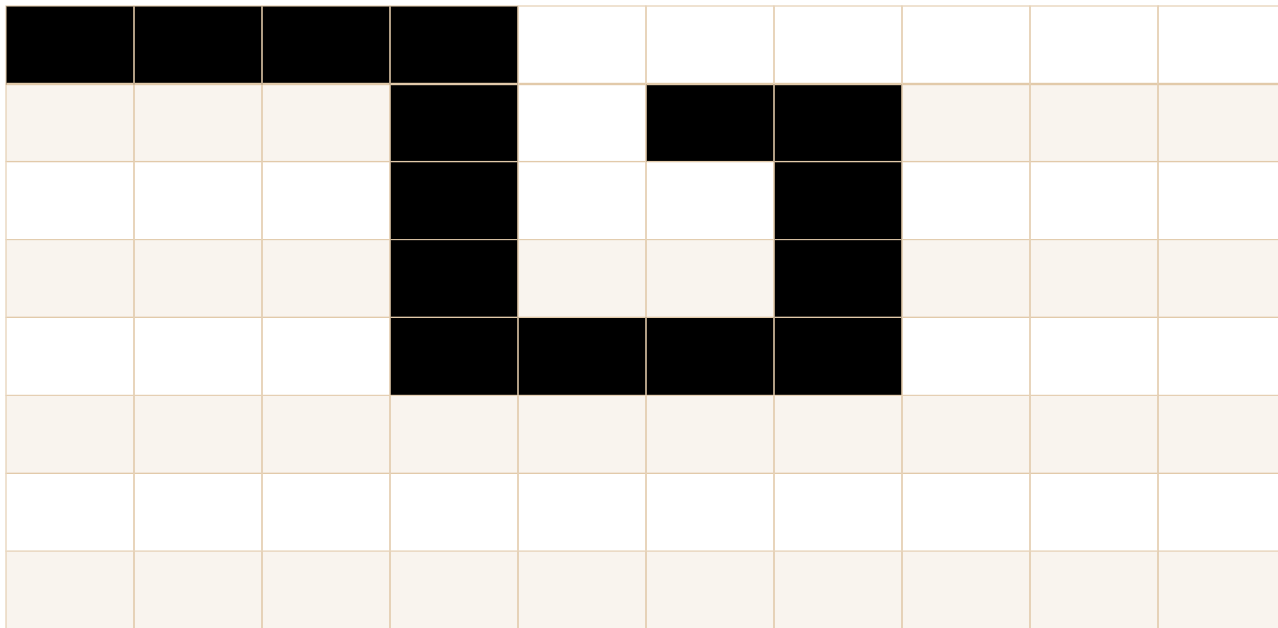


SAÍDA

Exemplo: mazes

VOLTA PARA ÚLTIMA SOLUÇÃO!

INÍCIO



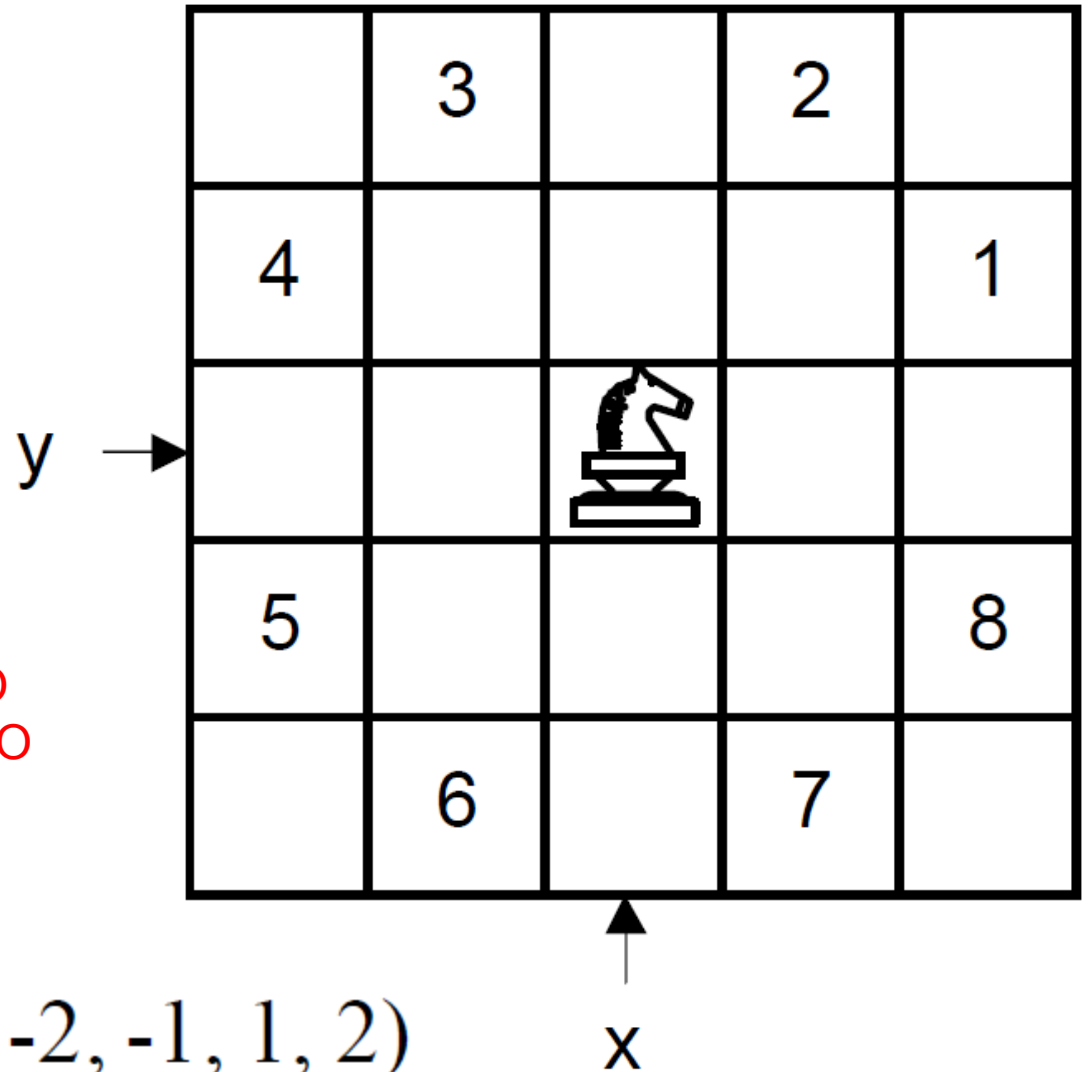
SAÍDA
DESCONHECIDA

O problema do cavalo no tabuleiro

Problema do cavalo no tabuleiro

- Suponha um tabuleiro de $n \times n$ campos
- O cavalo é colocado na posição inicial x_0, y_0
- Problema: encontrar uma série de $n^2 - 1$ movimentos no tabuleiro tal que cada campo seja visitado apenas uma vez.

POSIÇÕES POSSÍVEIS DO
CAVALO NO TABULEIRO DO
XADREZ



$$dx = (2, 1, -1, -2, -2, -1, 1, 2)$$

$$dy = (1, 2, 2, 1, -1, -2, -2, -1)$$


I-ÉSIMO
MOVIMENTO

POSIÇÕES
INICIAIS

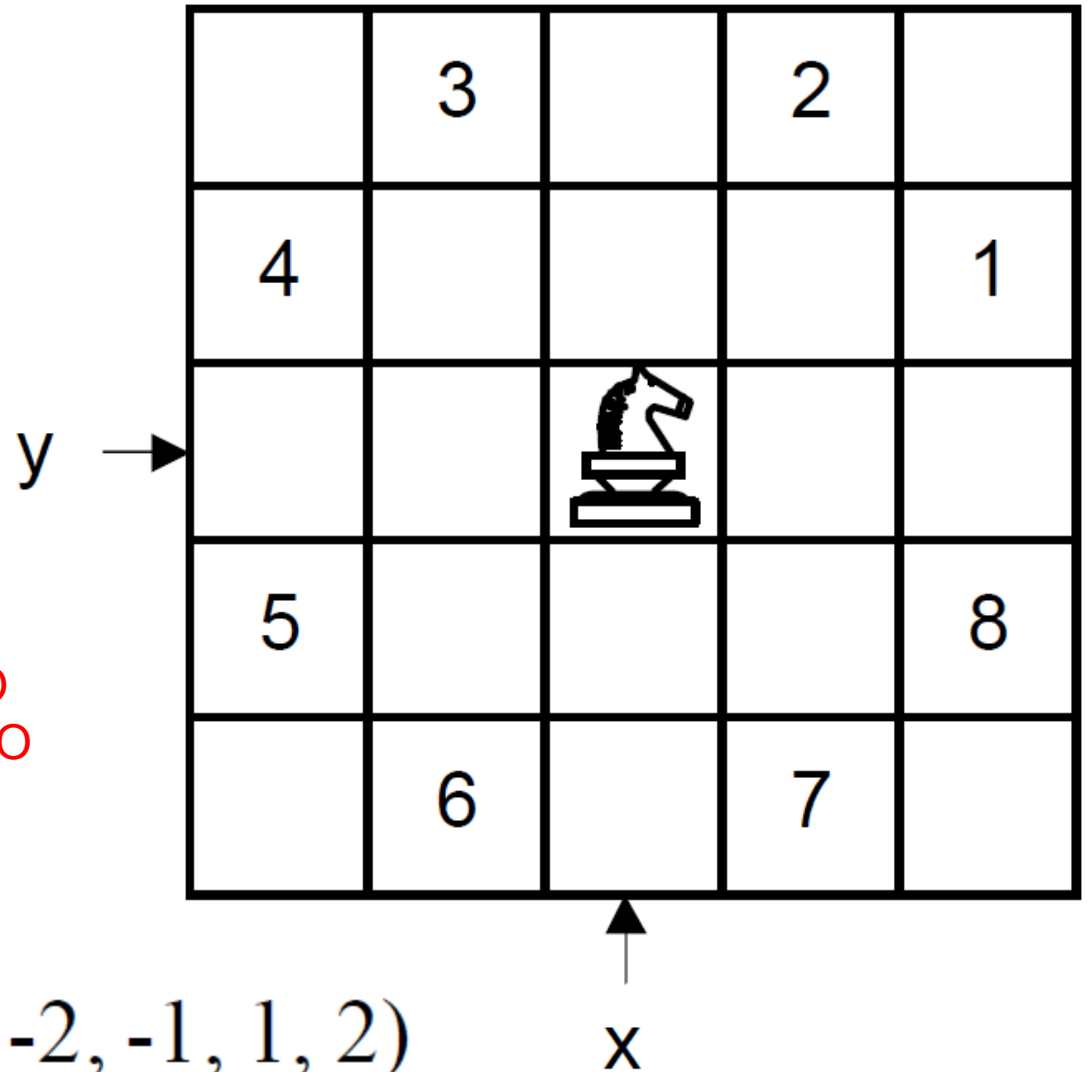
RETORNA Q=TRUE
CASO SUCESSO

```
PROCEDURE Try(i: INTEGER; x, y: index; VAR q: BOOLEAN);
  VAR u, v: INTEGER; q1: BOOLEAN;
BEGIN initialize selection of moves;
  REPEAT let <u,v> be the coordinates of the next move
    as defined by the rules of chess;
    IF (0 <= u) & (u < n) & (0 <= v) & (v < n) & (h[u,v] = 0) THEN
      h[u,v] := i;
      IF i < n*n THEN Try(i+1, u, v, q1);
      IF ~q1 THEN h[u,v] := 0 ELSE q1 := TRUE END
    END
  END
  UNTIL q1 OR no more candidates;
  q := q1
END Try
```

POSIÇÃO AINDA
NÃO VISITADA
 $h(u,v) == 0$



POSIÇÕES POSSÍVEIS DO
CAVALO NO TABULEIRO DO
XADREZ



$$dx = (2, 1, -1, -2, -2, -1, 1, 2)$$

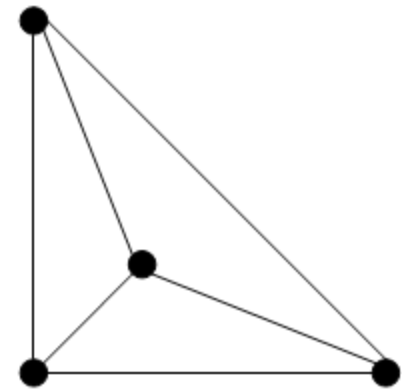
$$dy = (1, 2, 2, 1, -1, -2, -2, -1)$$

```
PROCEDURE Try(i, x, y: INTEGER; VAR q: BOOLEAN);
  VAR k, u, v: INTEGER; q1: BOOLEAN;
BEGIN k := 0;
  REPEAT k := k+1; q1 := FALSE;
    u := x + dx[k]; v := y + dy[k];
    IF (0 <= u) & (u < n) & (0 <= v) & (v < n) & (h[u,v] = 0) THEN
      h[u,v] := i;
      IF i < Nsq THEN Try(i+1, u, v, q1);
        IF ~q1 THEN h[u,v] := 0 END
      ELSE q1 := TRUE
      END
    END
  UNTIL q1 OR (k = 8);
  q := q1
END Try;
```

Busca em profundidade

Grafos

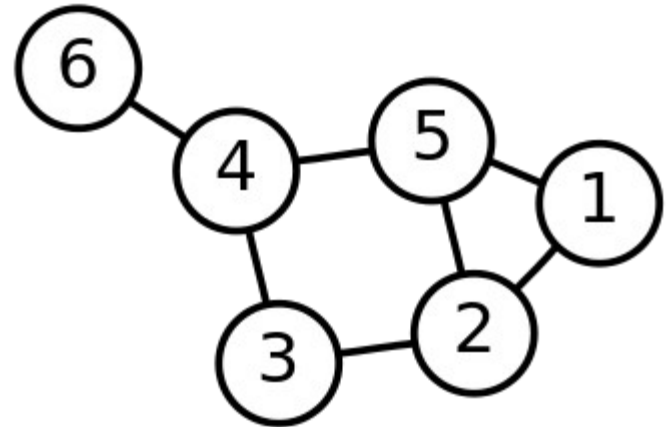
- Conjunto de vértices, ligados por arestas
 - Cidades conectadas por estradas
 - Páginas da Internet conectadas
 - Amigos do Facebook
 - E mais uma infinidade de sistemas reais ...



Representação

■ Matriz de Adjacências

- $M[1][2] = M[2][1] = 1$
- $M[1][5] = M[5][1] = 1$
- $M[2][5] = M[5][2] = 1$
- $M[2][3] = M[3][2] = 1$
- $M[3][4] = M[4][3] = 1$
- $M[4][5] = M[5][4] = 1$
- $M[4][6] = M[6][4] = 1$



Questão

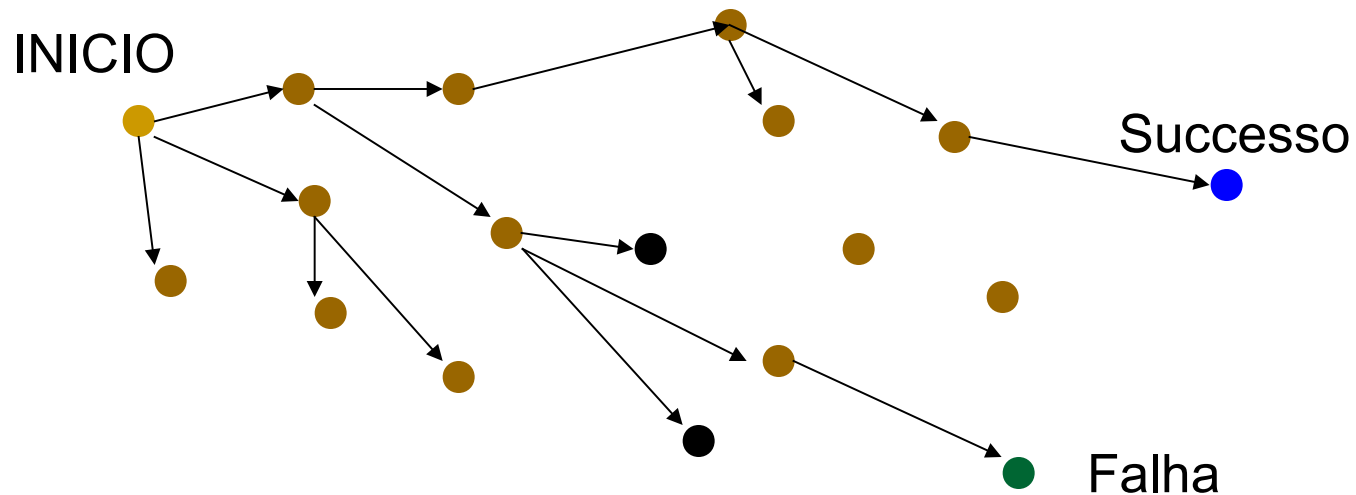
- Existe um caminho conectando dois vértices (src) e (tgt)?
 - Caso sejam vizinhos, qual o teste?
- E se não são vizinhos, posso reduzir ao mesmo subproblema?

Questão

- Existe um caminho conectando dois vértices (src) e (tgt)?
 - Caso sejam vizinhos, qual o teste?
- E se não são vizinhos, posso reduzir ao mesmo subproblema?
 - Ir para um dos vizinhos e aplicar novamente

Busca em profundidade

- Ir caminhando por vizinhos (em profundidade) até achar a solução ou achar um dead-end
- Se achar o dead-end volta na solução anterior
- **Atenção: ao caminhar, evitar ficar entrar em loops**




```
int dfsR( int M[N][N], int src, int tgt, int prev[] )
{
    int viz;    prev[src] = 1;
    if ( src == tgt ) return 1;
    for ( viz = 1; viz <= N; viz++ )
        if ( M[src][viz] != 0 && prev[viz] == -1 )
            if ( dfsR ( M, viz, tgt, prev ) ) return 1;

    prev[src] = -1;
    return 0;  }
```

Exercício

- Escreva um algoritmo que escreva o caminho utilizado na solução recursiva da busca em profundidade

```
int dfsR( int M[N][N], int src, int tgt, int prev[] )
{
    int viz;    prev[src] = 1;
    if ( src == tgt ) printf(src); return 1;
    for ( viz = 1; viz <= N; viz++ )
        if ( M[src][viz] != 0 && prev[viz] == -1 )
            if ( dfsR ( M, viz, tgt, prev ) == 1 )
                { printf(viz); return 1; }

    prev[src] = -1;
    return 0; }
```

Exercícios

Exercícios

- Escreva uma função recursiva que calcula a média dos elementos de um vetor.

Exercício

- Escreva uma função recursiva que encontre um dado elemento num vetor dividindo-o ao meio em cada chamada recursiva.
 - Este algoritmo é chamado busca binária

```
// Esta função recebe um vetor crescente v[e+1..d-1] e um
// número x tal que v[e] < x <= v[d] (portanto, e < d).
// Ela devolve um índice j em e+1..d tal que
// v[j-1] < x <= v[j].
```

```
int
bb( int x, int e, int d, int v[]) {
    if (e == d-1) return d;    // base da recursão
    else {
        int m = (e + d)/2;
        if (v[m] < x)
            return bb( x, m, d, v);
        else
            return bb( x, e, m, v);
    }
}
```

Recursão

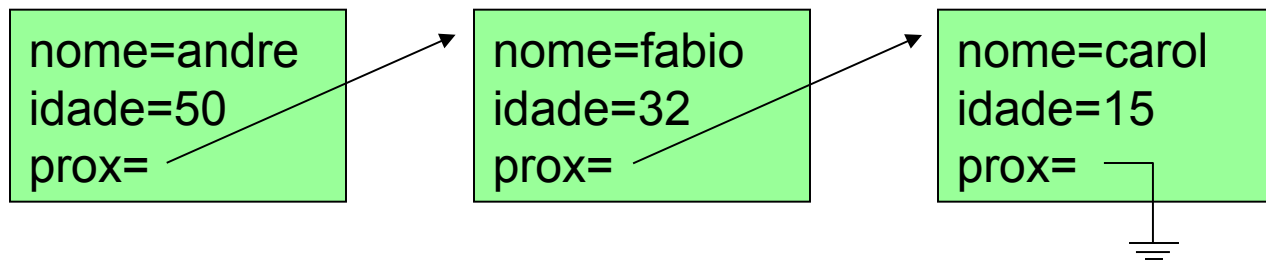
- Muito útil para lidar com estruturas de dados mais complexas
 - Listas sofisticadas, **árvores**, etc.

Exercício

- Imagine que você tem declarado vários blocos de memória para a seguinte estrutura

```
struct bloco {  
    char nome[50];  
    int idade;  
    struct bloco *prox;  
}
```

em que cada bloco aponta para o endereço do próximo bloco alocado. Por exemplo:



- Faça um programa que leia esses dados do usuário, armazenando-os nos blocos alocados dinamicamente, e, depois, chame uma função recursiva que imprima os dados armazenados na **ordem inversa**

Exercício

- Crie uma função recursiva para inserir uma célula no fim de uma lista

```
struct celula *insere (struct celula *cel,  
                      struct celula *ini);
```

```
//Aceita a célula a ser inserida e o inicio da lista
```

```
//Retorna o inicio da lista
```

Solução

```
struct celula *insere (struct celula *cel,  
                      struct celula *ini);  
  
{  
    cel->prox = NULL;  
    if ( ini == NULL ) return cel;  
    ini->prox = insere( cel, ini->prox );  
    return ini;  
}
```

Exercício

- Implementar a função para resolver o labirinto

```
int solveMaze ( const int *M, int N )
```

início = (0,0) e fim = (N-1,N-1)

Matriz de entrada contém 0's em posições vazias e 1's e blocos ocupados

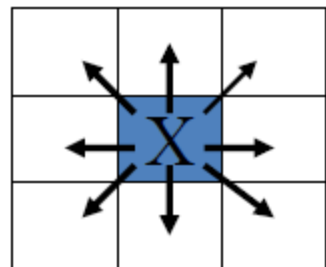
```
/* matriz que define o 'labirinto' */
```

```
int m[8][8];
```

```
/* dx e dy demarcam as 'células vizinhas' */
```

```
int dx[8] = {-1,-1,-1,0,0,1,1,1};
```

```
int dy[8] = {-1,0,1,-1,1,-1,0,1};
```



```
/* a função tenta() */
```

```
void tenta(int i, int j){
```

```
int k;
```

```
    if(m[i][j] == ' '){                /* alternativa é viável ?      */
```

```
        m[i][j] = 'x';                 /* marcar o caminho          */
```

```
        if((i==DI)&&(j==DJ))           /* atingiu o objetivo ?     */
```

```
            imprime();
```

```
        else {                          /* tentar todas alternativas */
```

```
            for(k=0; k < 8; k++) tenta(i+dx[k],j+dy[k]);
```

```
        }
```

```
        m[i][j] = ' ';                 /* desfazer a marca         */
```

```
    }
```

```
}
```

Exercício para casa

- Implemente a solução do Sudoku

Exercício para casa

- Implemente e mostre a “árvore de recursão” para um pequeno exemplo
 - Dado um arranjo v de n números inteiros, implemente uma função recursiva que retorne o maior elemento do arranjo
 - Considere como caso mais simples quando o arranjo possui dois elementos.
 - Considere como entrada tamanhos de vetores como potência de 2. Mostre que para um vetor contendo N elementos, $N-2$ chamadas recursivas são executadas, se cada chamada recursiva analisa metade do vetor.