

## Recurso

Prof<sup>a</sup>. Rose Yuri Shimizu

## Roteiro

- 1 Recurso
- 2 Recurso na programação

## Definição

- É a propriedade daquilo que pode se repetir várias vezes
- Dependência entre os elementos do conjunto
  - ▶ Elemento atual depende da determinação de um elemento anterior ou posterior
- Condição de parada: necessária para terminar a recursão
- **Exemplos de recursões matemáticas**

$$\text{Fatorial } n! = \begin{cases} 1, & \text{se } n = 0 \\ n.(n-1)!, & \text{se } n \geq 1 \end{cases}$$

$$\text{Fibonacci } f(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ f(n-1) + f(n-2), & \text{se } n \geq 2 \end{cases}$$

Roteiro

1 Recursão

2 Recursão na programação

## Algoritmos Recursivos

- São implementados através de funções:
  - ▶ Que invocam a si mesmos
  - ▶ Chamadas de funções recursivas
- Contribuem na implementação de algoritmos complexos em códigos mais compactos
- Sistemas atuais possibilitam uma execução eficiente das chamadas de função recursivas
  - ▶ Stacks: empilhamento das funções

## Algoritmos Recursivos

### Execução

- Comportamento de uma pilha
- Cada iteração: dados são empilhados, inclusive o endereço de quem chamou a função (para onde retornar)
- Última iteração:
  - ▶ Último invocado termina o seu processamento
  - ▶ É retirado da pilha e o topo da pilha retoma sua execução
- Processo de desempilhamento continua até a base da pilha
- Assim, o invocador inicial pode finalmente terminar seu processamento

## Algoritmos Recursivos

**Fatorial iterativo**  $n! = \begin{cases} 1, & \text{se } n = 0 \\ n.(n-1)!, & \text{se } n \geq 1 \end{cases}$

```
1 //fatorial iterativo
2 int n = 3;
3 int t = 1;
4 while (n>0) {
5     t *= n;
6     n--;
7 }
8 printf("%d! = %d\n", n, t); //n? t?
9
```

## Algoritmos Recursivos

**Fatorial recursivo**  $n! = \begin{cases} 1, & \text{se } n = 0 \\ n.(n-1)!, & \text{se } n \geq 1 \end{cases}$

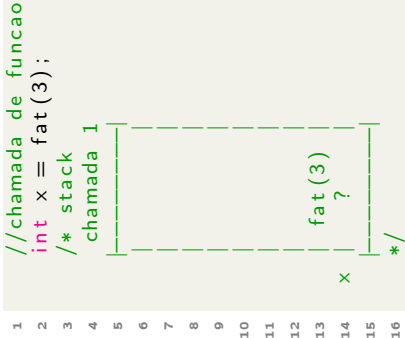
```
1 //fatorial recursivo
2 int fat(int n){
3     if (n==0) return 1;
4     return n * fat(n-1);
5 }
6
```

```
1 //chamada de funcao
2 int x = fat(3);
3
```



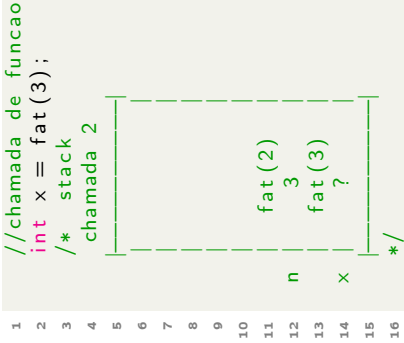
# Algoritmos Recursivos

```
1 //fatorial recursivo
2 int fat(int n){
3     if (n==0) return 1;
4     return n * fat(n-1);
5 }
6
```



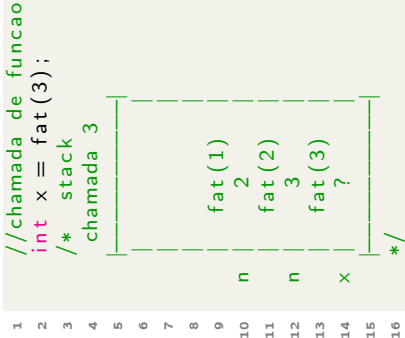
# Algoritmos Recursivos

```
1 //fatorial recursivo
2 int fat(int n){
3     if (n==0) return 1;
4     return n * fat(n-1);
5 }
6
```



# Algoritmos Recursivos

```
1 //fatorial recursivo
2 int fat(int n){
3     if (n==0) return 1;
4     return n * fat(n-1);
5 }
6
```



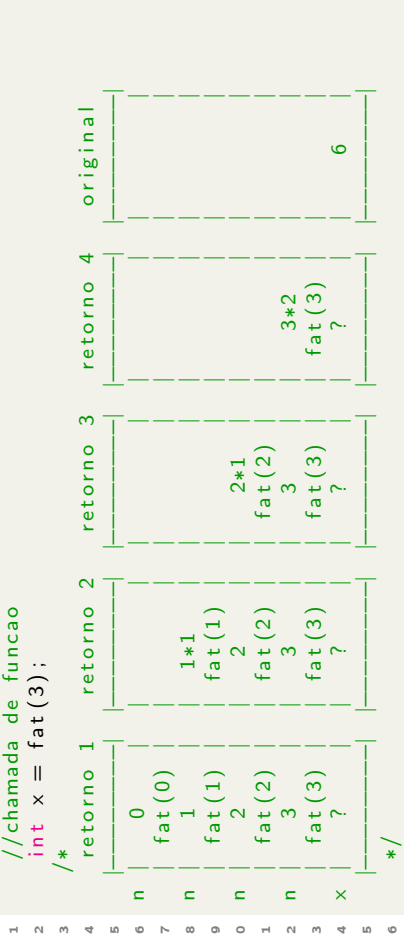
# Algoritmos Recursivos

```
1 //fatorial recursivo
2 int fat(int n){
3     if (n==0) return 1;
4     return n * fat(n-1);
5 }
```

```
1 //chamada de funcao
2 int x = fat(3);
3 /* stack
4 chamada 4
5
6
7 fat(0)
8 1
9 fat(1)
10 2
11 fat(2)
12 3
13 fat(3)
14 ?
15 x
16 */
```

# Algoritmos Recursivos

```
1 //fatorial recursivo
2 int fat(int n){
3     if (n==0) return 1;
4     return n * fat(n-1);
5 }
```



## Algoritmos Recursivos

$$\text{Fibonacci recursivo } f(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ f(n-1) + f(n-2), & \text{se } n \geq 2 \end{cases}$$

```
0 //fibonacci recursivo
1 int fib(int n){
2     if (n==0) return 0;
3     if (n==1) return 1;
4     return fib(n-1) + fib(n-2);
5 }
6
7 //chamada de funcao
8 int a = fib(3);
9
```

# Algoritmos Recursivos

## Validade dos algoritmos Recursivos

- A sequência recursiva precisa ser finita
- Podemos utilizar a indução matemática para provar sua validade
- Método da indução finita: provar propriedades que são verdadeiras para uma sequência de objetos
  - 1 Passo base  
(ex.:  $T$  é válido para  $n = 1$ )
  - 2 Passo indutivo ou hipótese da indução  
(ex.: para todo  $n > 1$ , se  $T$  é válido para  $n - 1$ , então  $T$  é válido para  $n$ )
- Podemos simplificar garantindo:
  - ▶ Caso base (condição de parada)
  - ▶ Que cada chamada decremente o valor da função recursiva, que garanta o alcance da condição de parada (garantindo a stack e o término da recursão)

## Algoritmos Recursivos

Exemplo da página do prof. Paulo Feofiloff

```
0 int max(int n, int v[]) {
1   if (n == 1) return v[0];
2   else {
3     int x = max(n-1, v);
4     // x is largest in v[0..n-2]
5
6     if (x > v[n-1]) return x;
7     else return v[n-1];
8   }
9 }
10 /* v[0..3] -> 77 88 66
11    max(3, v)
12    | max(2, v)
13    | | max(1, v)
14    | | | returns 77
15    | | | returns 88
16    | | | returns 88
17 */
```

Corretude por prova indutiva:

Passo base: para  $n = 1$ , o maior é  $v[0]$

Passo indutivo: para  $n \geq 1$ ,

- Sub-vetores menores de  $v$ :  
 $v[0..n-2]$  e  $v[n-1]$
- Se  $\text{max}(n-2, v)$  retorna o máximo de  $v[0..n-2]$
- Então, após a instrução  $x = \text{max}(n-1, v)$ ,  $x$  é o maior valor de  $v[0..n-2]$
- Portanto, a solução é o maior entre  $x$  e  $v[n-1]$  (outra parte do vetor)
  - ▶ Conforme computação da função
- Conclui-se então, que a função *max* retorna o maior elemento de um dado vetor



## Algoritmos Recursivos

Versão mais genérica da função max: intervalo do vetor

```
0 int max(int i, int n, int v[]) {  
1   if (i == (n-1)) return v[i];  
2   else {  
3       int x = max(i+1, n, v);  
4       //x is largest in v[i..n-1]  
5  
6       if (x > v[i]) return x;  
7       else return v[i];  
8   }  
9 }
```

## Algoritmos Recursivos

Exemplo do livro do Sedgewick

```
1 //resolver expressao matematica com notacao prefixa
2 // * + 7 * * 4 6 + 8 9 5 = (7+((4*6)*(8+9)))*5
3 char *a; int i=0;
4 int eval(){
5     int x=0;
6     while(a[i] == ' ') i++; //procura por operadores e digitos
7     if(a[i] == '+') {
8         i++;
9         return eval()+eval();
10    }
11    if(a[i] == '*') {
12        i++;
13        return eval()*eval();
14    }
15
16    //equivalente numerico de uma sequencia de caracteres
17    while((a[i] >= '0') && (a[i] <= '9'))
18        //calcula o decimal, centena ... + valor numerico
19        x = 10*x + (a[i++] - '0'); //tabela ascii
20
21    return x;
22 }
```

## Algoritmos Recursivos

```
1 resolver expressao matematica com notacao prefixa
2 * + 7 * * 4 6 + 8 9 5 = (7+((4*6)*(8+9)))*5
3
4 |eval() * + 7 * * 4 6 + 8 9 5
5 | |eval() + 7 * * 4 6 + 8 9
6 | | |eval() 7
7 | | |eval() * * 4 6 + 8 9
8 | | | |eval() * 4 6
9 | | | | |eval() 4
10 | | | | |eval() 6
11 | | | | |return 4 * 6 = 24
12 | | | | |eval() + 8 9
13 | | | | |eval() 8
14 | | | | |eval() 9
15 | | | | |return 8 + 9 = 17
16 | | | | |return 24 * 17 = 408
17 | | | | |return 7 + 408 = 415
18 | | | | |eval() 5
19 | | | | |return 5*415 = 2075
20
21
```

# Algoritmos Recursivos

## Analise

Exemplo do livro do Sedgewick

```
int puzzle(int n) {  
    if (n==1) return 1;  
    if (n%2 == 0) {  
        return puzzle(n/2);  
    } else {  
        return puzzle(3*n+1);  
    }  
}
```

```
//aumenta o recurso aumentando o custo da função  
puzzle(3*n+1)
```

## Usos da recursividade – recomendações

- Se uma instância for pequena: use força bruta, resolva diretamente
- Senão, reduza em instância menores do mesmo problema
- Resolva por partes e volte para instância original
- Essa é a técnica da “divisão e conquista”
  - ▶ Resolva os subproblemas para resolver o problema
  - ▶ Consiste em:
    - ★ Dividir o problema em partes menores
    - ★ Encontrar as soluções das partes
    - ★ Combinando-as para obter a solução global (conquista)

```
0  divisao_conquista(d) {  
1      se simples  
2          calculo_direto(d)  
3      senao  
4          combina(divisao_conquista(decompoe(d)))  
5      }
```

- ▶ O custo computacional geralmente é determinada pela relação da recorrência (profundidade da pilha)
- ▶ Tende a algoritmos mais eficientes
- ▶ Auxilia em problemas mais complexos, dividindo em problemas menores
- ▶ Facilita a paralelização na fase da conquista

## Usos da recursividade - recomendações

```
0 int max(int a[], int l, int r){  
1     int u, v;  
2     int m = (l+r)/2;  
3     u = max(a, l, m);  
4     v = max(a, m+1, r);  
5     if (u>v) return u;  
6     return v;  
7 }  
8
```

## Usos da recursividade - recomendações

- Pode ser aplicado em problemas de:
  - ▶ Operações de multiplicação de matrizes
  - ▶ Planejamento de caminhos em robotica movel
  - ▶ Problemas de tentativa e erro (*backtracking*: errou? volta e tenta outra solução)
  - ▶ Compiladores (analísadores léxicos)
  - ▶ **Manipulação das estrutura de dados** (formas de armazenamento de dados)
  - ▶ **Algoritmos de pesquisas, ordenação**