

Introdução à Ciência da Computação

Disciplina: 113913

Professor: Luiz Augusto F. Laranjeira
luiz.laranjeira@gmail.com

Universidade de Brasília – UnB
Campus Gama

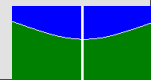


10a. FUNÇÕES RECURSIVAS



Função Recursiva

- É uma função que se chama a si mesma. A idéia é usar a própria função (que se está a definir) em sua definição.
- Em toda função recursiva existe:
 - a) Um *passo básico* cujo resultado é imediatamente conhecido.
 - b) Um *passo recursivo* em que se tenta resolver um sub-problema do problema inicial.
- Um erro comum ao se escrever uma função recursiva é a recursão nunca parar. Isto normalmente ocorre porque:
 - a) O caso mais simples (passo básico) não é detectado; e/ou
 - b) A recursão (passo recursivo) não diminui a complexidade do problema.



Fatorial - Função Iterativa

- Uma função iterativa que calcula o fatorial de um número natural ***n***.
- Definição de fatorial de um número:
 - $n! = n * (n-1) * (n-2) * \dots * 2 * 1$
 - $0! = 1$

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



Fatorial - Função Recursiva

- Uma função recursiva que calcula o fatorial de um número natural ***n***.
- Definição de fatorial de um número:
 - $n! = n * (n-1) * (n-2) * \dots * 2 * 1$
 - $0! = 1$
 - $n! = n * (n-1)!$ (definição recursiva)

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



Fatorial Recursivo - Execução

■ FATORIAL RECURSIVO:

$\text{fat}(5) = 5 * \text{fat}(4)$; equação 1

$\text{fat}(4) = 4 * \text{fat}(3)$; equação 2

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

$\text{fat}(2) = 2 * \text{fat}(1)$; equação 4

$\text{fat}(1) = 1 * \text{fat}(0)$; equação 5

$\text{fat}(0) = 1$; equação 6

Substituindo a equação 6 na equação 5 temos: $\text{fat}(1) = 1 * 1$

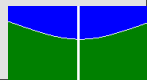
Substituindo a equação 5 na equação 4 temos: $\text{fat}(2) = 2 * 1$

Substituindo a equação 4 na equação 3 temos: $\text{fat}(3) = 3 * 2$

Substituindo a equação 3 na equação 2 temos: $\text{fat}(4) = 4 * 6$

Substituindo a equação 2 na equação 1 temos: $\text{fat}(5) = 5 * 24$

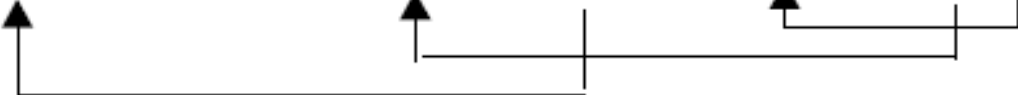
E assim **$\text{fat}(5) = 120$**



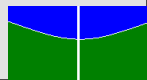
Fatorial Recursivo - Visualização

- Visualmente podemos analisar o fatorial de 3, $\text{fat}(3)$, da seguinte forma:

Chamada	1ª	2ª	3ª	4ª
N	3	2	1	0
Fat	$3 * \text{Fat}(2)$	$2 * \text{Fat}(1)$	$1 * \text{Fat}(0)$	1
	$3 * 2 = 6$	$2 * 1 = 2$	$1 * 1 = 1$	



- Para cada nova chamada da função, mais memória no stack é utilizada para guardar as informações (variáveis) daquela chamada.



Alocação de Memória de Programa (Run-Time)

ALOCAÇÃO DE MEMÓRIA

1. Variáveis globais
 - Inicializadas: **data**
 - Não-inicializadas: **bss**
2. Variáveis estáticas: **bss**
3. Variáveis locais: **stack**
4. Argumentos e endereços de retorno de chamadas de funções: **stack (pilha)**
5. Alocação dinâmica (malloc/free): **heap (monte)**
6. Código executável do programa: **text**

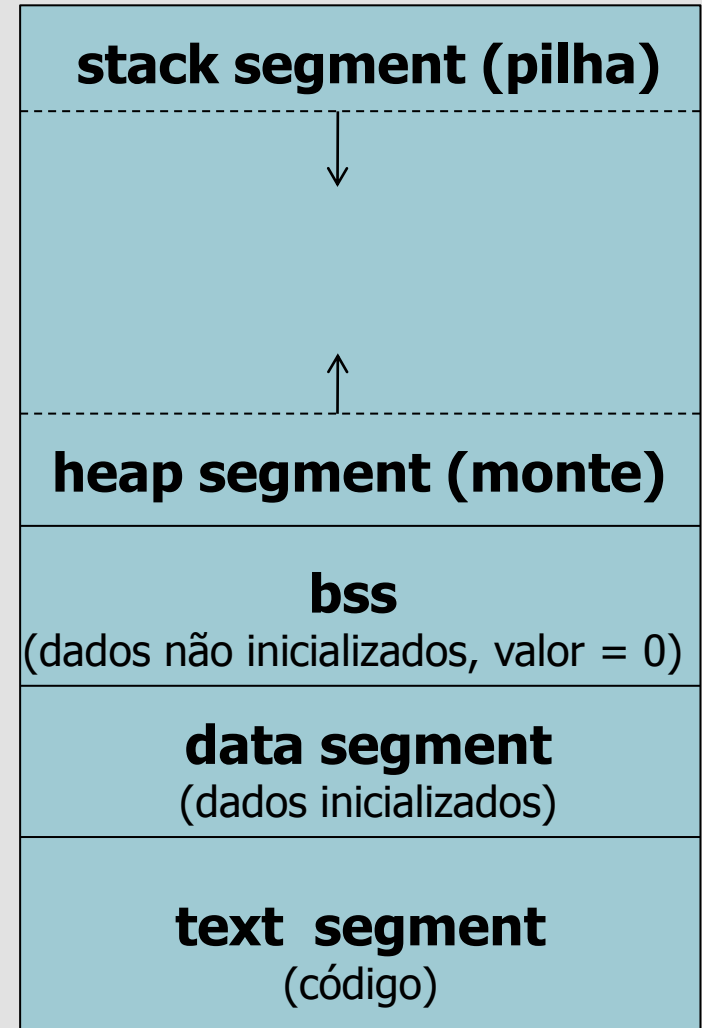
bss = **b**lock **s**tarted by **s**ymbol

bss: nome de uma operação existente no assembler do IBM 704 (mid 1950's).

endereços
mais altos



endereços
mais baixos



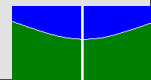
Função Recursiva - Cuidados

- 1) Temos sempre que ter um ponto de parada (passo básico), para não entrarmos em um loop infinito, por exemplo na função recursiva fatorial, $\text{fat}(0) = 1$.
- 2) Observar que o passo recursivo deve diminuir a complexidade do problema, também para evitar um loop infinito.
- 3) Analisar se a função realizará um número muito grande de chamadas a ela mesma, pois a recursividade gasta um tempo significativo para empilhar (guardar na memória chamada pilha ou stack) esses valores e depois os desempilhar (recuperar e liberar memória).



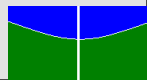
Recursividade – Observações (1)

- 1) Uma solução recursiva ocupa mais memória e é mais lenta que a solução iterativa para o mesmo problema.
- 2) Em cada instância, todos os parâmetros e variáveis locais são criados novamente (no stack), independentemente dos que já existiam antes.
- 3) A cada nova chamada, o sistema deve guardar a posição de memória de onde a chamada foi feita, para que posteriormente possa voltar ao lugar certo.
- 4) Uma função recursiva é mais elegante e menor que a sua versão iterativa, além de exibir com maior clareza o processo utilizado, desde que o problema ou dados sejam naturalmente definidos através da recorrência.



Recursividade – Observações (2)

- 5) É fácil criar funções que chamam a si mesmas.
- 6) É difícil reconhecer situações apropriadas para a utilização de recursividade.
- 7) Há certos problemas cuja natureza permite uma solução recursiva bem mais simples e intuitiva do que a solução iterativa.



Exercício 1

Escreva uma função recursiva em C que compute $T(n)$ para a sequência T que se segue:

- $T(1) = 1$
- $T(n) = T(n-1) + 3$ para $n \geq 2$

Escreva um programa que calcule o termo de ordem N desta sequência. O programa deverá executar um loop que pede o valor de n e escreve na tela o valor do termo de ordem n da seguinte forma:

$$T(21) = 345 \quad (\text{para } n = 21)$$

O programa deve terminar quando o usuário entrar um valor de n que seja ≤ 0 .



Exercício 1 - Solução

```
#include <stdio.h>
#include <stdlib.h>

int SeqT(int n)
{
    if (n == 1) return(1);
    else return(SeqT(n-1) + 3);
}

int main() {

    int N=0;

    // Inicializando a tela
    system ("cls");

    // Publicando o que o origrama faz
    printf("\n\nEste programa calcula o termo de
           ordem n, T(n), de uma sequencia T.\n");
    printf("(Para sair do programa digite um valor de
           n que seja menor que 1)\n\n");
```

```
do {
    printf("  Digite o valor de n  ( n>=1 ): ");
    scanf("%d", &N);
    if (N >= 1) printf("\n      T(%d) = %d\n\n",
                      N, SeqT(N));
} while (N >= 1);  /* fim-do-while */

// Pulando linhas para clareza na tela
printf("\n\n\n");

// Fazendo o programa esperar antes de sair
system("pause");

return(0);

} // end of main
```



Exercício 2

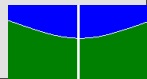
Escreva uma função recursiva em C que compute $S(n)$ para a sequência S que se segue:

$$p, p - q, p + q, p - 2q, p + 2q, p - 3q, \dots$$

Escreva um programa que calcule o termo de ordem N desta sequência. O usuário deverá entrar primeiramente com os valores de p e q e, em seguida, o programa deverá executar um loop que pede o valor de N e escreve na tela o valor do termo de ordem N da seguinte forma:

$$S(21) = 345 \quad (\text{para } n = 21)$$

O programa deve terminar quando o usuário entrar um valor de N que seja ≤ 0 .



Exercício 2 - Resolução (1)

Escrever uma função recursiva em C que compute $S(n)$ para a sequência S que se segue:

$$p, p - q, p + q, p - 2q, p + 2q, p - 3q, \dots$$

$$S(1) = p$$

$$S(2) = p - q$$

$$S(3) = p + q$$

$$S(4) = p - 2q$$

$$S(5) = p + 2q$$

$$S(n) = ? \quad \text{para } n > 1$$



Exercício 2 – Resolução (2)

Escreva uma função recursiva em C que compute $S(n)$ para a sequência S que se segue:

$$p, p - q, p + q, p - 2q, p + 2q, p - 3q, \dots$$

$$S(1) = p$$

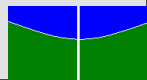
$$S(2) = p - q$$

$$S(3) = p + q$$

$$S(4) = p - 2q$$

$$S(5) = p + 2q$$

$$S(n) = S(n-1) + (-1)^{n-1} \cdot (n-1)q \quad \text{para } n > 1$$



Exercício 2 – Solução

```
int  p=M, q=N;
int  Seq (int n)
{
    if (n == 1) return(p) ;
    if (n%2 == 0) return(Seq(n-1) - (n-1)*q) ;
    else return(Seq(n-1) + (n-1)*q) ;
}
```

// O teste para verificar que o valor de n é ≥ 1 dever ser
// feito no programa principal e não na função.

