



# Algoritmos e Estrutura de Dados Avançado

Encontro 04 – Recursividade



# Materiais e dúvidas



**Nenhum** material será enviado via e-mail.  
Os materiais serão disponibilizados no **AVA** e no disco virtual: [bit.ly/edauniderp](https://bit.ly/edauniderp)



Dúvidas, questionamentos, entre outros  
deverão ser realizados **APENAS** pelo  
**e-mail** e **AVA**.



Para ingressar no grupo do **Whatsapp** da  
disciplina acesse o link [linklist.bio/noiza](https://linklist.bio/noiza) e  
selecione sua disciplina.

# Recursividade

- Recursividade é a capacidade de um programa (função ou procedimento) fazer uma ou mais chamadas a si mesmo.
- Na execução de um programa recursivo, uma pilha é responsável pelo armazenamento das variáveis recursivas.
- Uma função recursiva tem que seguir duas regras básicas:
  - Ter uma condição de parada (para garantir que uma chamada recursiva não criará um loop infinito)
  - Tornar o problema mais simples

# Recursividade

- Genericamente um módulo recursivo segue o algoritmo abaixo:

**SE <condição de parada satisfeita>**

**Retornar**


**senão**

**Divida o problema num caso mais simples  
utilizando recursão**


# Decomposição do Fatorial(3)

$$\text{Fatorial}(3) = 3 * \text{Fatorial}(2)$$

$$\text{Fatorial}(3) = 3 * 2 * 1 = 6$$


$$\text{Fatorial}(2) = 2 * \text{Fatorial}(1)$$

$$\text{Fatorial}(2) = 2 * 1 = 2$$


$$\text{Fatorial}(1) = 1 * \text{Fatorial}(0)$$

$$\text{Fatorial}(1) = 1$$


$$\text{Fatorial}(0) = 1$$

# Projeto para implementar funções recursivas

- Toda função recursiva possui dois elementos:
  - Resolver parte do problema (caso base) e
  - Reduzir o tamanho do problema (caso geral).

No caso do nosso exemplo,  $\text{Fatorial}(0)$  é o caso base

# Regras para projetar uma função recursiva

- Determinar o caso base;
- Determinar o caso geral;
- Combinar o caso base e o caso geral na função.
- **Atenção:** Cada chamada da função deve reduzir o tamanho do problema e movê-lo em direção do caso base. O caso base deve terminar sem chamar a função recursiva, isto é, executar um return.

# Solução Iterativa (exemplo01)

// Solução iterativa para o cálculo do fatorial

```
long fatorial(int n){  
    // declarações locais  
    long factN = 1;  
    int i;  
    for (i = 1; i <= n; i++)  
        factN *= i;  
    return factN;  
} //fim fatorial
```



# Solução Recursiva (exemplo02)

// Solução recursiva para o cálculo do fatorial

```
long fatorial(int n){
```

```
    if (n == 0)
```

```
        return 1;
```

```
    else
```

```
        return (n*fatorial(n-1));
```

```
} //fim fatorial
```

# Números de Fibonacci

- Números de Fibonacci são uma série na qual cada número é a soma dos dois números anteriores:

1, 1, 2, 3, 5, .....

- Para iniciar a série é preciso conhecer os dois primeiros números.

# Generalização da série de Fibonacci

- Dados (caso base):
  - $\text{Fibonacci}_1 = 1;$
  - $\text{Fibonacci}_2 = 1;$
- Caso geral:
  - $\text{Fibonacci}_n = \text{Fibonacci}_{n-1} + \text{Fibonacci}_{n-2}$
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

# Série de Fibonacci (exemplo03)

- Escreva uma função que determine a série de Fibonacci com n termos.

```
long fib(int n){  
    if (n == 1 || n==2) // caso base  
        return 1;  
    else  
        return (fib(n-1) + fib(n-2)); // caso geral  
}
```

# Análise da função Fibonacci

- Ineficiência em Fibonacci
  - Termos  $\text{Fib}_{n-1}$  e  $\text{Fib}_{n-2}$  são computados independentemente
  - Número de chamadas recursivas = número de Fibonacci!
  - Custo para cálculo de  $\text{Fib}_n$ 
    - Exponencial!!!

# Fibonacci não recursivo (exemplo04)

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

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

# Limitações da recursão

- Soluções recursivas podem envolver alto overhead devido à chamada de funções;
- A cada chamada da função recursiva, espaço de memória (na pilha) é alocada. Se o número de chamada recursiva é muito grande, pode não ter espaço suficiente na pilha para executar o programa (*segment fault*)

**IMPORTANTE:** Cada chamada recursiva de uma função em C aloca novos espaços de memória para todas as variáveis locais definidas pela função, inclusive para aquelas definidas como parâmetros da função.

# Limitações da recursão

- As funções para determinar o fatorial e a série de Fibonacci são melhores implementadas por funções iterativas;

**Isto significa que soluções iterativas são sempre melhores que as recursivas????**

**NÃO: ALGUNS ALGORITMOS SÃO MAIS FÁCEIS DE IMPLEMENTAR USANDO RECURSÃO E SÃO MAIS EFICIENTES!**



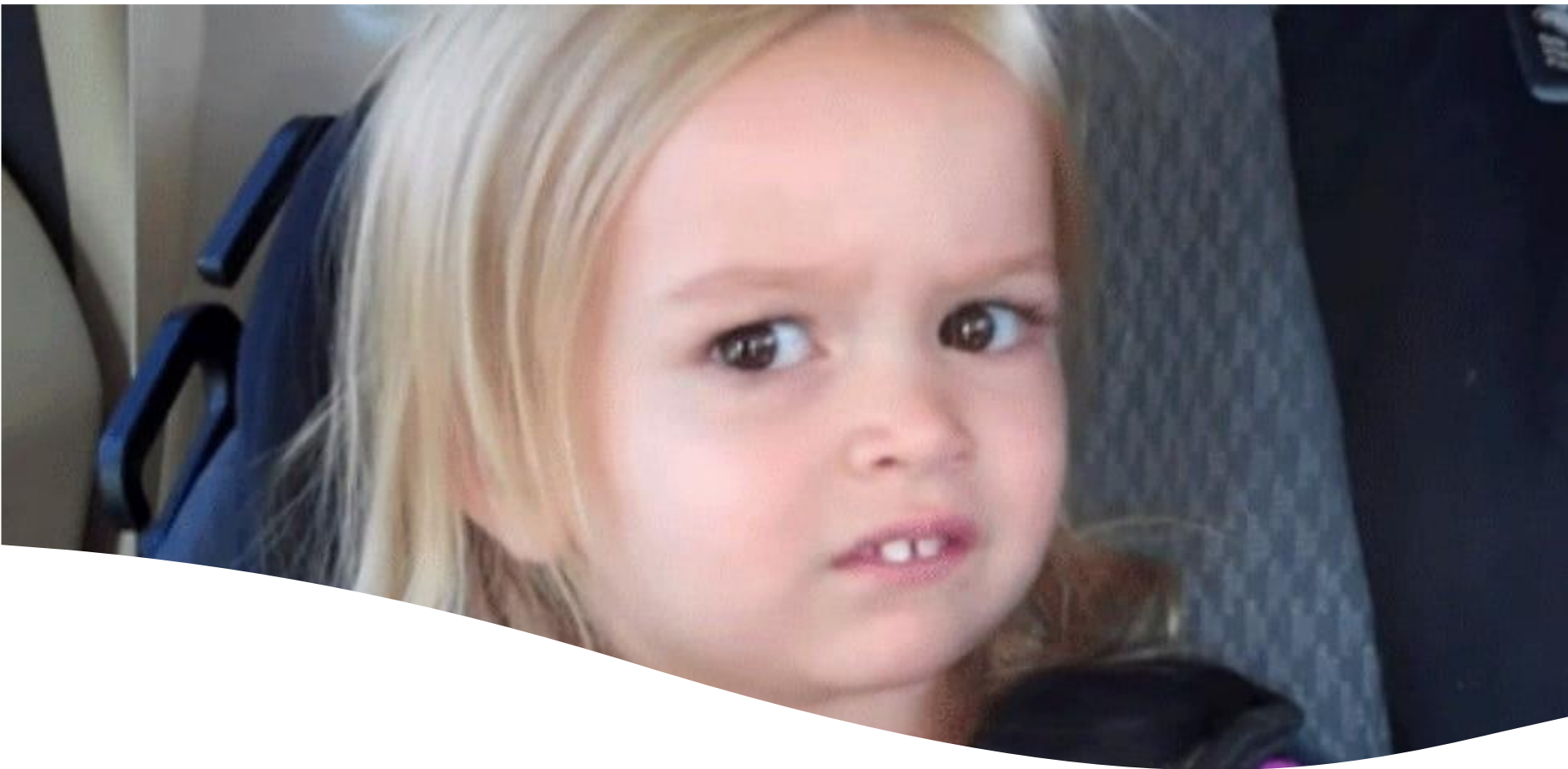
# Vantagens e Desvantagens

- **Vantagens da recursão**

- Redução do tamanho do código fonte
- Maior clareza do algoritmo para problemas de definição *naturalmente recursiva*

- **Desvantagens da recursão**

- Baixo desempenho na execução devido ao tempo para gerenciamento das chamadas
- Dificuldade de depuração dos subprogramas recursivos, principalmente se a recursão for muito profunda



# Atividade

Lista de Exercícios



uniderp

# Exercícios

- Baseado no seu conhecimento adquirido e em pesquisas a serem realizadas desenvolva a Lista03.
- Prepare seus exercícios no ambiente Dev C++ usando Linguagem C.
- O trabalho pode ser realizado de forma individual ou em duplas.
- Os arquivos deverão ser apresentados até o final da aula.
- Data de entrega: 27/08/2024.



# Encerramento



**DÚVIDAS, CRÍTICAS E SUGESTÕES, ENVIAR PARA:**

**[noiza@anhanguera.com](mailto:noiza@anhanguera.com)**



uniderp