

# INF 112 – Programação 2

## Aula: Recursividade



# Droste

HAARLEM - HOLLAND



## cacao

Netto 250 g

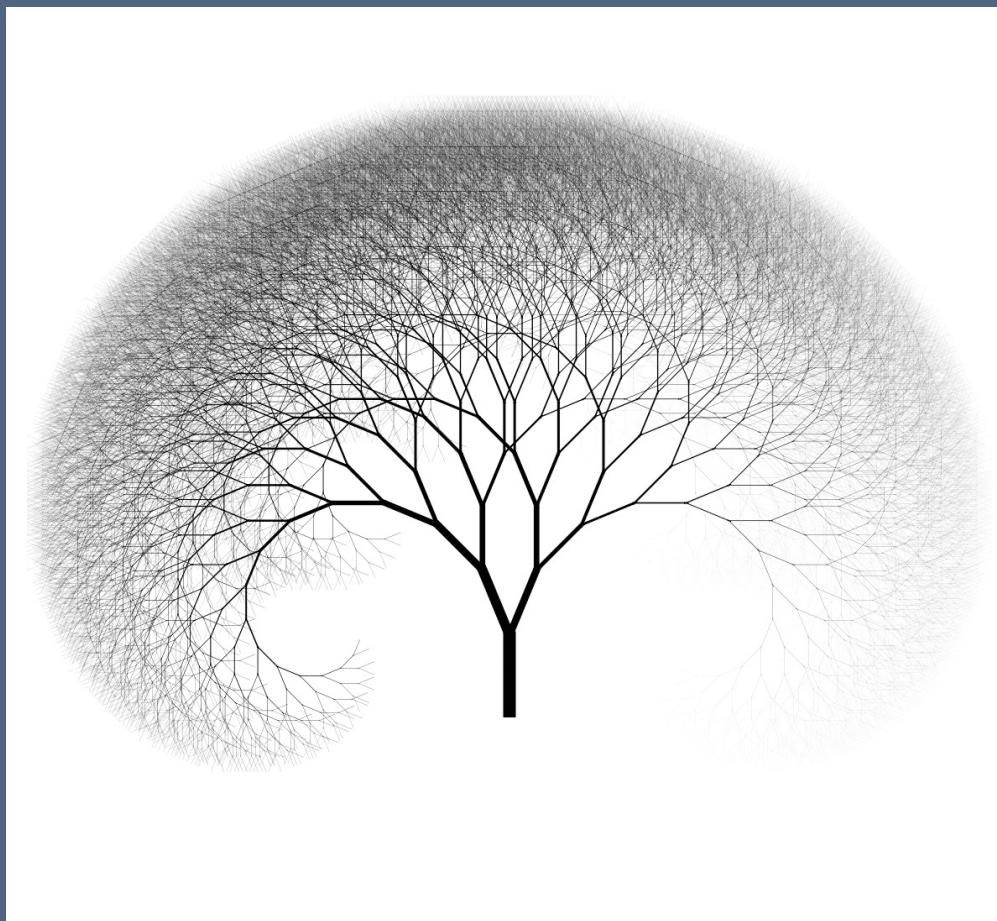
# Algoritmo para aprender recursão

# Algoritmo para aprender recursão

Para aprender recursão, você primeiro precisa aprender recursão.

# Recursividade

Conceito muito utilizado na computação, matemática, etc..  
Também encontrado com frequência na natureza



# Recursividade

Na computação, uma função é recursiva quando ela é definida em termos de si mesma.

# Recursividade

A recursividade é tão importante na computação que muitas siglas famosas são acrônimos recursivos.

Ex:

- GNU: “GNU is Not Unix”
- Wine: “Wine Is Not an Emulator”.

# Recursividade

Exemplo: fatorial

O fatorial de  $n$  pode ser calculado de forma iterativa (não recursiva) com base na seguinte definição:

$$n! = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1$$

Como seria uma implementação de um código para cálculo de fatorial?

# Recursividade

```
int fat(int n) {  
    int resp = 1;  
    for(int i = n; i >= 2; i--)  
        resp *= i;  
    return resp;  
}
```

# Recursividade

Normalmente o fatorial é definido na matemática de forma recursiva:

$$0! = 1$$

$$n! = n \times (n-1)!$$

Esse tipo de relação é chamada de “relação de recorrência”: o fatorial é definido em termos de si próprio, mas há um caso base utilizado para “parar” a avaliação da função.

# Recursividade

Exemplo de implementação recursiva do fatorial:

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

Exercício: rastreie a execução desse código supondo que se deseja calcular o fatorial do número 4.

Note que o “if” é de extrema importância nesse código, já que ele serve para fazer as chamadas recursivas pararem no caso base.

# Recursividade – Pilha de Execução

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

n = 4  
return = ?

fat (4)

# Recursividade – Pilha de Execução

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

n = 3  
return = ?

fat (3)

n = 4  
return = ?

fat (4)

# Recursividade – Pilha de Execução

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

n = 2  
return = ?

n = 3  
return = ?

n = 4  
return = ?

fat (2)

fat (3)

fat (4)

# Recursividade – Pilha de Execução

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

n = 1  
return = ?

n = 2  
return = ?

n = 3  
return = ?

n = 4  
return = ?

fat (1)

fat (2)

fat (3)

fat (4)

# Recursividade – Pilha de Execução

n = 0  
return = 1

n = 1  
return = ?

n = 2  
return = ?

n = 3  
return = ?

n = 4  
return = ?

fat (0)

fat (1)

fat (2)

fat (3)

fat (4)

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

# Recursividade – Pilha de Execução

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

n = 1  
return =  $1 * 1$

fat (1)

}

n = 2  
return = ?

fat (2)

n = 3  
return = ?

fat (3)

n = 4  
return = ?

fat (4)

# Recursividade – Pilha de Execução

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

n = 2  
return =  $2 * 1$

n = 3  
return = ?

n = 4  
return = ?

fat (2)

fat (3)

fat (4)

# Recursividade – Pilha de Execução

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

n = 3  
return =  $3 \times 2$

n = 4  
return = ?

fat (3)

fat (4)

# Recursividade – Pilha de Execução

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

n = 4  
return = 4\*6

← fat (4)

# Recursividade

## Vantagens e Desvantagens

```
int fat(int n) {  
    int resp = 1;  
    for(int i = n; i >= 2; i--)  
        resp *= i;  
    return resp;  
}
```

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

# Recursividade

Vantagens: código elegante; mais simples de entender; por ser mais simples, talvez ele pode ser otimizado pelo programador com maior facilidade utilizando algumas técnicas (que serão vistas mais para frente).

Desvantagens: eficiência (as chamadas de função são **empilhadas** e realizadas várias vezes), pode consumir mais **memória**; requer mais habilidade do programador para não fazer...

# Recursividade

Como assim uso de memória? Não estou alocando nada!

Errado!

# Recursividade

## Exercícios:

- Escreva uma função recursiva que conta quantos dígitos um número positivo tem.
- Escreva uma função recursiva que recebe um arranjo de números inteiros (e o tamanho do arranjo) e retorne o produtório de tais números
- Escreva uma função recursiva que recebe como parâmetro dois números (a,b) e, então, retorna o somatório dos números inteiros entre a e b (inclusive).

# Recursividade

Como poderíamos converter/imprimir um número **positivo** em binário?

Os dígitos binários podem ser extraídos tirando-se o resto da divisão do número por 2 e, então, dividindo-o por 2...

Ao dividir um número binário por 2, eliminamos o bit menos significativo do número.

# Recursividade

Rastreie o código para n = 22

```
void printBinIterativo(int n) {  
    while(n != 0) {  
        cout << n % 2;  
        n/=2;  
    }  
}
```

# Recursividade

Rastreie o código para n = 22

```
void printBinIterativo(int n) {  
    while(n != 0) {  
        cout << n % 2;  
        n/=2;  
    }  
}
```

Há um problema no código... qual é?

# Recursividade

Solução que imprime o número do bit mais significativo para o menos significativo:

```
void printBinIterativo(int n) {
    int temp[32];
    int numDigitos = 0;
    while(n != 0) {
        temp[numDigitos] = n % 2;
        n /= 2;
        numDigitos++;
    }
    for(int i = numDigitos - 1; i >= 0; i--)
        cout << temp[i];
}
```

# Recursividade

Como seria um código recursivo para imprimir um número inteiro em binário?

# Recursividade

Como seria um código recursivo para imprimir um número inteiro em binário?

```
void printBinRecursivo(int n) {  
    if (n == 0)  
        return;  
    printBinRecursivo(n / 2);  
    cout << n % 2;  
}
```

**Exercício:** Rastreie o código para  $n = 22$

# Recursividade

**Exercício:** implemente um código recursivo que, dados um arranjo de caracteres contendo um número binário (com até 31 dígitos) e o número de dígitos desse número, retorne um número inteiro positivo contendo o valor decimal do número.

# Recursividade

```
int binary_to_decimal(char *num, int n) {  
    if (n == 0)  
        return (num[0] == '1'? 1: 0);  
    return 2 * binary_to_decimal(num, n - 1) +  
           (num[n] == '1'? 1: 0);  
}  
  
...  
  
binary_to_decimal(num, strlen(num) - 1)
```

# Recursividade

Outro exemplo de recursividade...

Impressão de todos os subconjuntos de **n** itens.

Como seria um código para imprimir todos os subconjuntos?

# Recursividade

```
void printCombinacoes(bool estados[], int n, int begin) {  
    if (begin == n) {  
        for(int i=0; i < n; i++)  
            cout << estados[i];  
        cout << endl;  
    } else {  
        estados[begin] = false;  
        printCombinacoes(estados, n, begin + 1);  
        estados[begin] = true;  
        printCombinacoes(estados, n, begin + 1);  
    }  
}
```