

Funções



Prof. André Backes | @progdescomplicada

Função

- Funções são blocos de código que podem ser nomeados e chamados de dentro de um programa.
 - **print()**: função que escreve na tela
 - **input ()**: função que lê o teclado

Função

- Facilitam a estruturação e reutilização do código.
 - Estruturação: programas grandes e complexos são construídos bloco a bloco.
 - Reutilização: o uso de funções evita a cópia desnecessária de trechos de código que realizam a mesma tarefa, diminuindo assim o tamanho do programa e a ocorrência de erros

Função | Estrutura

- Toda função começa com o comando **def**, seguido de um nome associado a aquela função, **nomefunção**
- Forma geral de uma função

```
def nomefunção(lista-de-parâmetros):  
    instrução 1  
    instrução 2  
    ...  
    instrução n
```

Função | Corpo

- O corpo da função é a sua alma.
 - É formado pelos comandos que a função deve executar
 - Ele processa os parâmetros (se houver), realiza outras tarefas e gera saídas (se necessário)

```
def imprimeCompras():  
    compras = ["Miojo", "Ovo", "Leite", "Pão"]  
    print("Lista de compras")  
    for item in compras:  
        print("Produto: ", item)  
# fim da função  
  
print("Antes da função")  
  
imprimeCompras()  
  
print("Depois da função")
```

Função | Corpo

- De modo geral, evita-se fazer operações de leitura e escrita dentro de uma função
 - Uma função é construída com o intuito de realizar uma tarefa específica e bem-definida
 - As operações de entrada e saída de dados (**input ()** e **print()**) devem ser feitas em quem chamou a função
 - Isso assegura que a função construída possa ser utilizada nas mais diversas aplicações, garantindo a sua generalidade

Função | Ordem de Execução

- Ao chamar uma função, o programa que a chamou é pausado até que a função termine a sua execução

```
def imprimeCompras():  
    compras = ["Miojo", "Ovo", "Leite", "Pão"]  
    print("Lista de compras")  
    for item in compras:  
        print("Produto: ", item)  
# fim da função  
  
print("Antes da função")  
  
imprimeCompras()  
  
print("Depois da função")
```

Função | Parâmetros

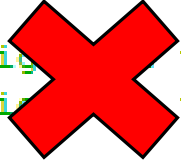
- A lista de parâmetros é uma lista de variáveis
 - nome1, nome2, ... , nomeN
 - Pode-se definir quantos parâmetros achar necessários
- Pode-se deixar os parênteses vazios se a função não recebe nenhum parâmetro de entrada

```
def soma(x, y):  
  
def imprime():
```


Função | Parâmetros

- É por meio dos parâmetros que uma função recebe informação do programa principal (isto é, de quem a chamou)
 - Não é preciso fazer a leitura das variáveis dos parâmetros dentro da função
 - Na chamada abaixo, o parâmetro x recebeu o valor 5 enquanto o parâmetro y recebeu o valor de z

```
def soma(x, y):  
    x = int(input("Digite o valor de X: "))  
    y = int(input("Digite o valor de Y: "))  
# fim da função  
  
z = int(input("Digite o valor de Z: "))  
soma(5, z)
```



Função | Parâmetros com valor padrão

- Podemos definir **valores padrão** para parâmetros da função
 - Isso faz com que aquele parâmetro se torne **opcional**, ou seja, se não for definido o **valor padrão** será usado
 - Parâmetros com **valores padrão** tem de vir sempre no final

Função | Parâmetros com valor padrão

Exemplo

```
def reajuste(salario, juros = 0.25):  
    return salario + salario * juros  
  
print("Reajuste 1: ",reajuste(100))  
print("Reajuste 2: ",reajuste(100,0.10))
```

Saída

```
>>>  
Reajuste 1: 125.0  
Reajuste 2: 110.0  
>>>
```

Função | Retorno

- Uma função pode ou não retornar um valor
 - Se ela retornar um valor, alguém deverá receber este valor.
 - O valor retornado pela função é dado pelo comando **return**

Função | Retorno

Função sem retorno

```
def imprimeCompras():  
    compras = ["Miojo", "Ovo", "Leite", "Pão"]  
    print("Lista de compras")  
    for item in compras:  
        print("Produto: ", item)  
# fim da função  
  
print("Antes da função")  
  
imprimeCompras()  
  
print("Depois da função")
```

Função com retorno

```
import math  
def volumeEsfera(r):  
    V = 4/3 * math.pi * r**3  
    return V  
# fim da função  
  
x = volumeEsfera(1.0)  
print(x)
```

Função | Retorno

- Uma função pode ter mais de uma declaração **return**.
 - Quando o comando **return** é executado, a função termina imediatamente
 - Todos os comandos restantes são **ignorados**

Função | Retorno

Exemplo

```
def maior(x,y):  
    if x > y:  
        return x  
    else:  
        return y  
    print("Essa mensagem não será impressa!");  
  
# fim da função  
  
x = int(input("Digite o valor de X: "))  
y = int(input("Digite o valor de Y: "))  
z = maior(x,y)  
print("O maior valor é :",z)
```

Saída

```
>>>  
Digite o valor de X: 2  
Digite o valor de Y: 7  
O maior valor é : 7
```

Função | Declaração

- Funções devem ser declaradas antes de serem utilizadas

```
x = int(input("Digite o valor de X: "))
y = int(input("Digite o valor de Y: "))
z = maior(x,y)
print("O maior valor é :",z)

def maior(x,y):
    if x > y:
        return x
    else:
        return y
    print("Essa mensagem não será impressa!");

# fim da função

Digite o valor de X: 2
Digite o valor de Y: 4
Traceback (most recent call last):
  File "D:\Aulas UFU\Material Teórico - UFU\Python\teste_funcoes.py", line 72,
<module>
    z = maior(x,y)
NameError: name 'maior' is not defined
```


Função | Declaração

- Uma função criada pelo programador pode utilizar qualquer outra função
 - Inclusive as que foram criadas pelo próprio programador

```
def soma(valores):  
    s = 0  
    for x in valores:  
        s = s + x  
    return s  
  
# fim da função  
  
def media(valores):  
    return soma(valores)/len(valores)  
  
# fim da função  
  
print("Soma = ", soma(range(1, 5)))  
print("Média = ", media([1, 2, 3, 4]))
```

Variáveis dentro da função

- Funções estão sujeitas ao escopo das variáveis
 - Uma variável definida no programa e sem indentação é **global**.
 - Ela pode ser acessada em qualquer lugar do programa ou função

```
def func():  
    print("Função = ",x)  
  
x = 10  
print("Antes = ",x)  
func()  
print("Depois = ",x)  
  
>>>  
Antes = 10  
Função = 10  
Depois = 10  
>>>
```

Variáveis dentro da função

- Uma variável **global** pode ser acessada em qualquer lugar do programa ou função, mas não pode ser alterada pela função
 - Ao tentar alterar uma variável **global** o que ocorre é a criação de uma variável **local** que ofusca completamente a variável **global**

```
def func():  
    x = 20  
    print("Função = ",x)  
  
x = 10;  
print("Antes = ",x)  
func()  
print("Depois = ",x)  
  
>>>  
Antes = 10  
Função = 20  
Depois = 10  
>>>
```

Variáveis dentro da função

- Para atribuir um novo valor a uma variável global precisamos utilizar o comando **global**
 - Isso faz com que a variável manipulada dentro da função seja a do escopo **global**

```
def func():  
    global x  
    x = 20  
    print("Função = ",x)  
  
x = 10;  
print("Antes = ",x)  
func()  
print("Depois = ",x)  
>>>  
Antes = 10  
Função = 20  
Depois = 20  
>>>
```

Variáveis dentro da função

- Variáveis definidas **dentro da função** (com indentação ou parâmetros) somente podem ser acessadas dentro da função, nunca fora dela

```
import math
def volumeEsfera(r):
    V = 4/3 * math.pi * r**3
    return V
# fim da função

x = volumeEsfera(1.0)
print(x)
print(V)

>>>
4.1887902047863905
Traceback (most recent call last):

  print(V)
NameError: name 'V' is not defined
```

Passagem de Parâmetros

- Em várias linguagens de programação, o tipo de passagem de parâmetros define se as modificações realizadas nos parâmetros **dentro** da função irão se refletir **fora** da função
- Na linguagem Python, os parâmetros de uma função podem ou não ser modificado, sendo definidos como **mutáveis** e **imutáveis**

Passagem de Parâmetros

- Na verdade, sempre que passamos um parâmetro para a função, estamos passando a referência a um objeto via **atribuição**

```
def func(N1,lista1):  
    #comandos da função  
    print("Teste função")  
  
N = 10  
lista = [1,2,3,4]  
  
func(N,lista) # N1 = N, lista1 = lista
```

Passagem de Parâmetros

- Parâmetros imutáveis
 - O conteúdo/valor do parâmetro é modificado dentro da função via atribuição.
 - Isso gera um novo objeto
 - Mesmo que esse valor mude dentro da função, nada acontece com o valor de fora da função

```
def incrementa(N):  
    print('valor = ',N)  
    N = N + 1  
    print('valor = ',N)
```

```
x = 1  
incrementa(x)  
print('valor = ',x)
```

Saída

```
valor = 1  
valor = 2  
valor = 1
```


Passagem de Parâmetros

- Parâmetros mutáveis
 - O conteúdo/valor do parâmetro é modificado dentro da função sem usar a operação de atribuição
 - Isso **não** gera um novo objeto
 - Nesse caso, alterar o parâmetro pode influenciar no “valor” da variável fora da função

```
def soma(valores):  
    s = 0  
    for x in valores:  
        s = s + x  
  
    valores.append("João");  
    return s  
  
# fim da função  
lista = [1, 2, 3, 4]  
print("Soma = ",soma(lista))  
print("Último elemento: ",lista[len(lista)-1])  
  
>>>  
Soma = 10  
Último elemento: João  
>>>
```

Recursão

Recursão

- Em Python, uma função pode chamar qualquer outra função
 - Por exemplo, dentro de qualquer função que nós criarmos é possível chamar a função **print()** ou **input()**, ou qualquer função definida pelo programador
- Uma função também pode chamar a si própria
 - A qual chamamos de **função recursiva**

Recursão

- A recursão também é chamada de definição circular. Ela ocorre quando algo é definido em termos de si mesmo.
- Um exemplo clássico de função que usa recursão é o cálculo do fatorial de um número:
 - $3! = 3 * 2!$
 - $4! = 4 * 3!$
 - $n! = n * (n - 1)!$

Recursão

$$0! = 1$$

$$1! = 1 * 0!$$

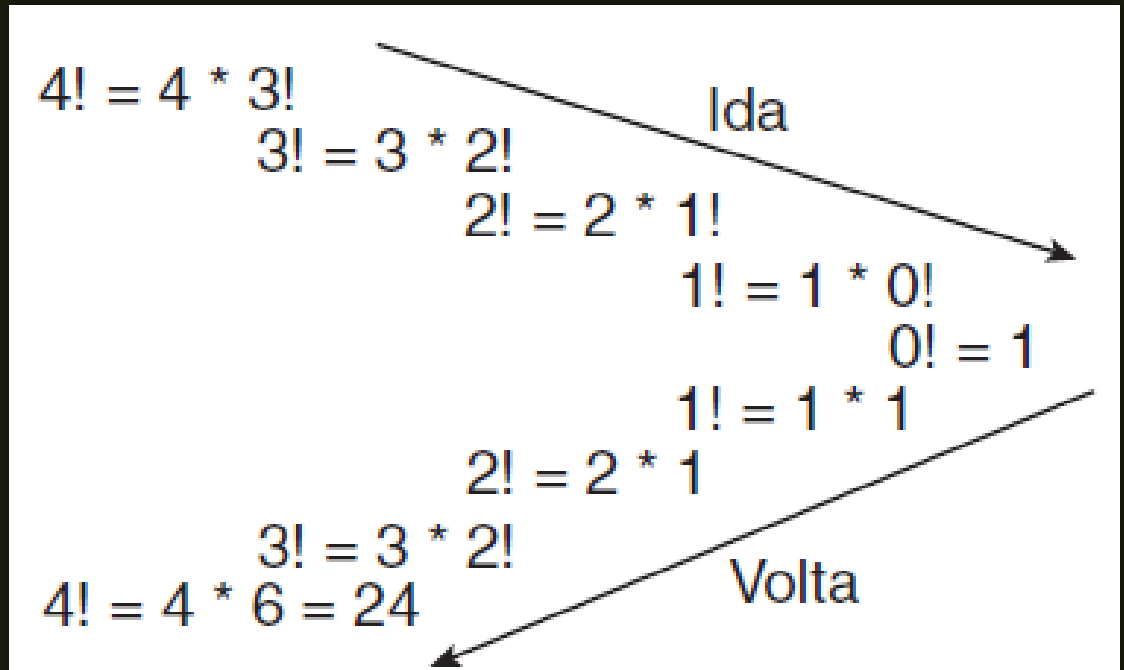
$$2! = 2 * 1!$$

$$3! = 3 * 2!$$

$$4! = 4 * 3!$$

$$n! = n * (n - 1)! : \text{fórmula geral}$$

$$0! = 1 : \text{caso-base}$$



Recursão

- Em geral, formulações recursivas de algoritmos são frequentemente consideradas "mais enxutas" ou "mais elegantes" do que formulações iterativas
- Porém, algoritmos recursivos tendem a necessitar de mais espaço de memória do que algoritmos iterativos

Recursão

Sem recursão

```
def fatorial(N):  
    fat = 1  
    for i in range(1,N+1):  
        fat = fat * i  
    return fat  
  
# fim da função  
  
x = int(input("Digite o valor de N: "))  
y = fatorial(x)  
print("O fatorial de ",x," é ",y)
```

Com recursão

```
def fatorial(N):  
    if N == 0:  
        return 1  
    else:  
        return N * fatorial(N-1)  
  
# fim da função  
  
x = int(input("Digite o valor de N: "))  
y = fatorial(x)  
print("O fatorial de ",x," é ",y)
```

Recursão

- Todo cuidado é pouco ao se fazer funções recursivas
 - **Critério de parada:** determina quando a função deverá parar de chamar a si mesma
 - O parâmetro da chamada recursiva deve ser sempre modificado, de forma que a recursão chegue a um término

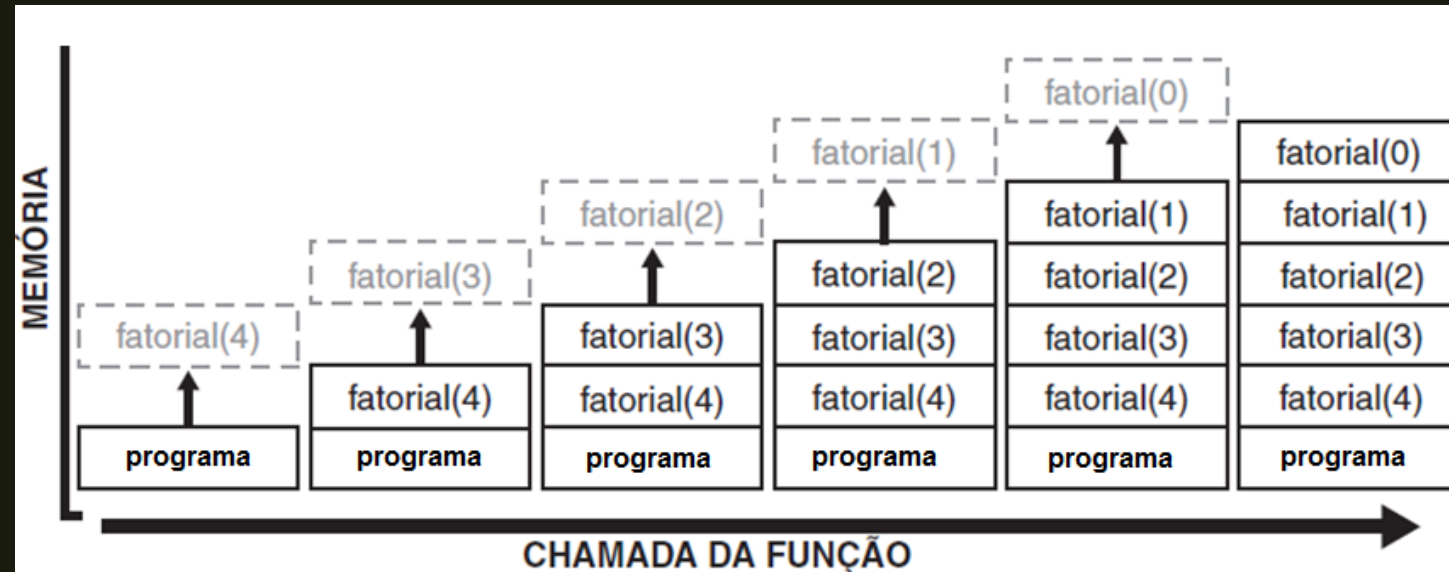
Recursão

- Exemplo: cálculo do fatorial

```
def fatorial(N) :  
    if N == 0: #critério de parada  
        return 1  
    else:  
        return N * fatorial(N-1) # parâmetro sempre muda  
  
# fim da função  
  
x = int(input("Digite o valor de N: "))  
y = fatorial(x)  
print("O fatorial de ",x," é ",y)
```

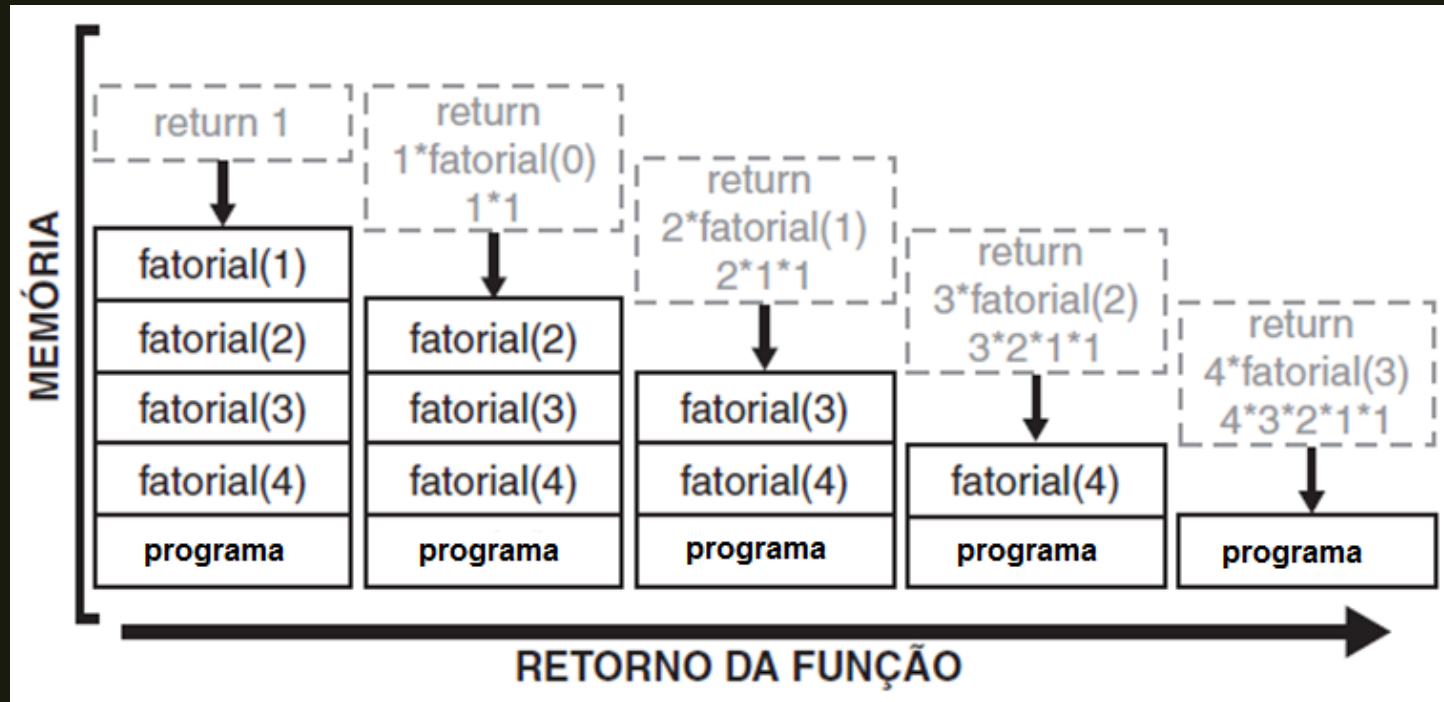
Recursão

- O que acontece na chamada da função fatorial com um valor como $N = 4$?
 - $y = \text{fatorial}(4)$



Recursão

- Uma vez que chegamos ao caso-base, é hora de fazer o caminho de volta da recursão



Recursão | Fibonacci

- Essa sequência é um exemplo clássico de recursão
 - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- Sua solução recursiva é muito mais elegante e simples...

Recursão | Fibonacci

Sem recursão

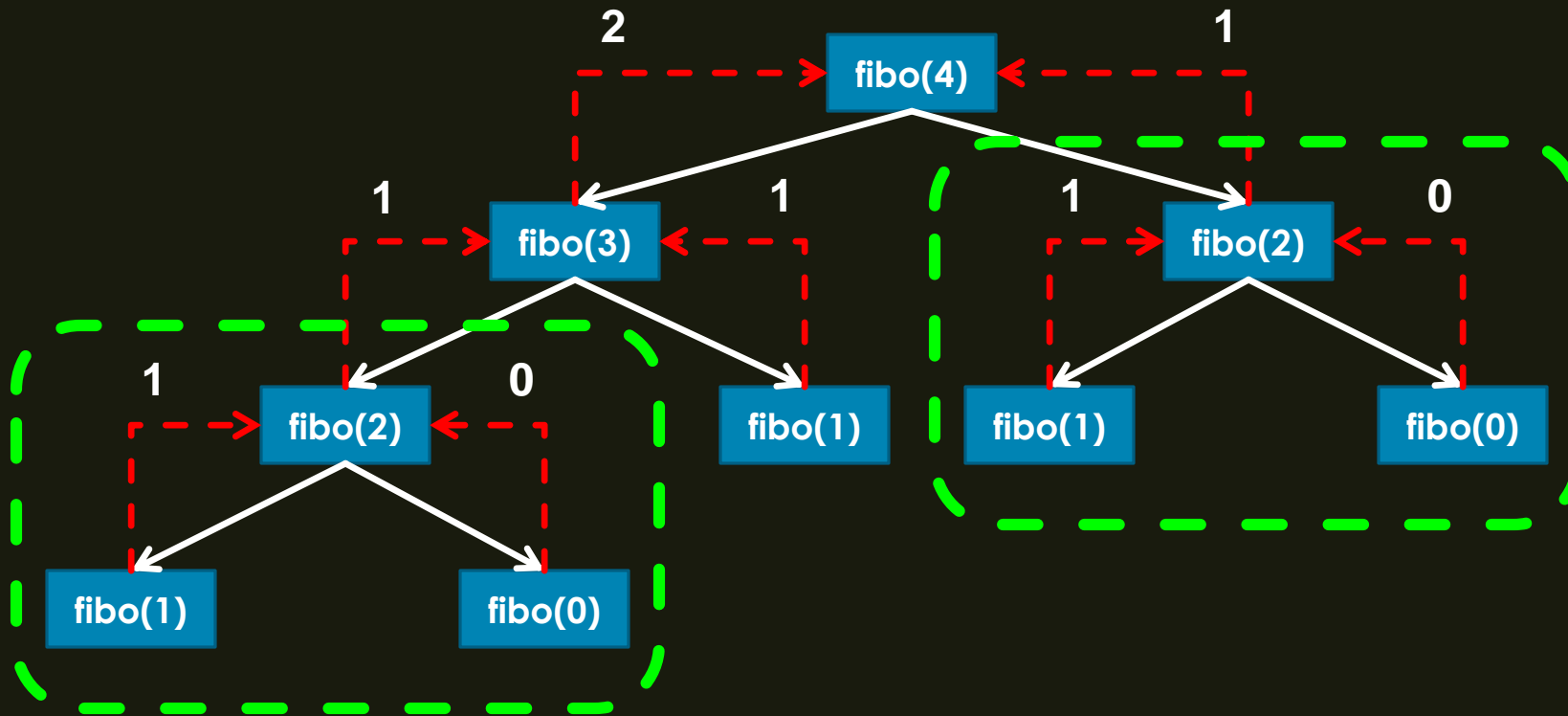
```
def fibonacciI(N):  
    if N == 0 or N == 1:  
        return N  
    else:  
        A = 0  
        B = 1  
        cont = 1  
        while cont < N:  
            C = A + B  
            cont = cont + 1  
            A = B  
            B = C  
        return C
```

Com recursão

```
def fibonacciR(N):  
    if N == 0 or N == 1:  
        return N  
    else:  
        return fibonacciR(N-1) + fibonacciR(N-2)
```

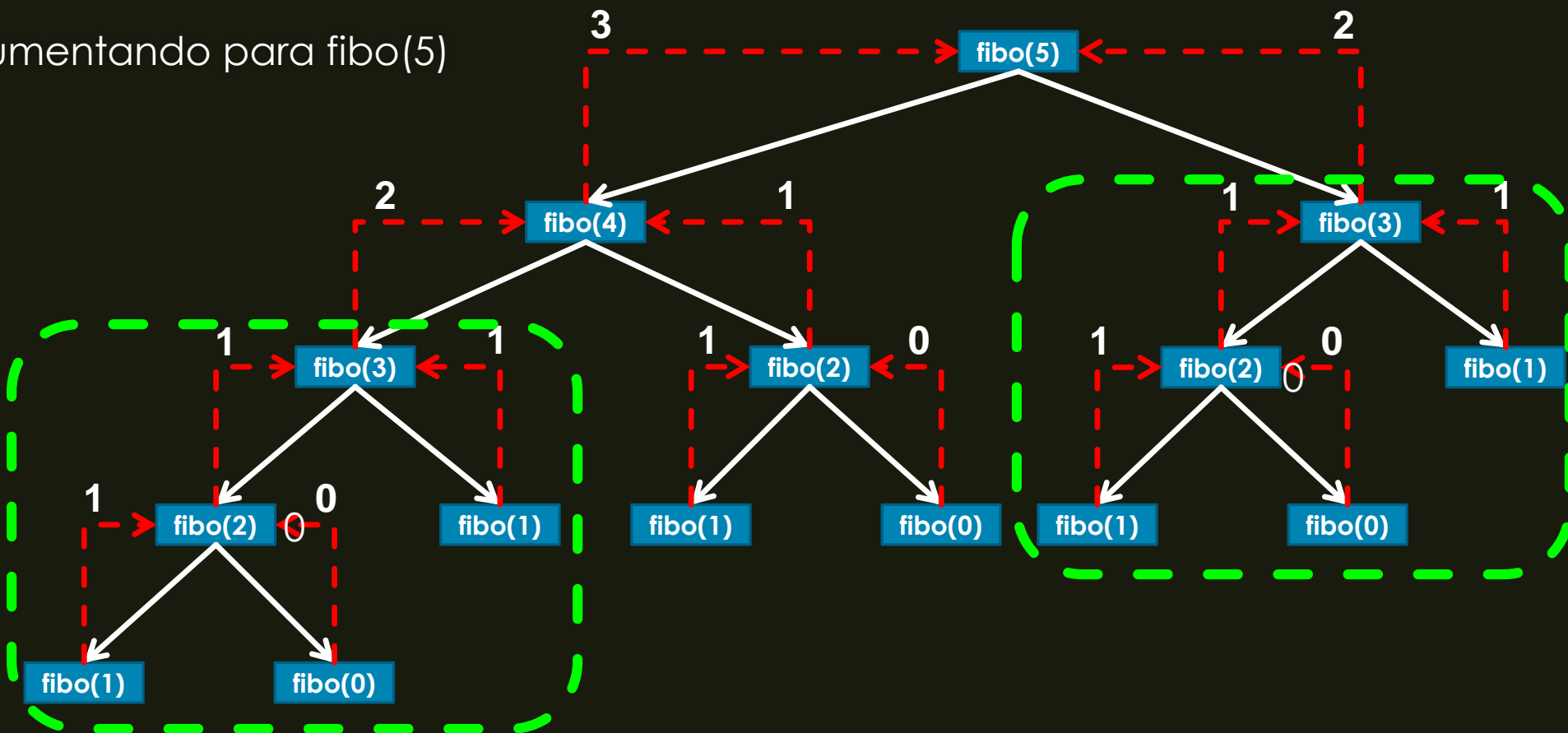
Recursão | Fibonacci

- ... mas como se verifica na imagem, elegância não significa eficiência



Recursão | Fibonacci

- Aumentando para fibo(5)



Recursão | Fibonacci

- Comparação de tempo de execução

```
x = int(input("Digite o valor de N: "))
t0 = time.process_time();
y = fibonacciI(x)
t1 = time.process_time();
z = fibonacciR(x)
t2 = time.process_time();
print("Fibonacci Iterativo: %.10f" % (t1-t0))
print("Fibonacci Recursivo: %.10f" % (t2-t1))
```

```
Digite o valor de N: 35
Fibonacci Iterativo: 0.0000000000
Fibonacci Recursivo: 3.1406250000
```


Material Complementar

- Vídeo Aulas
 - Aula 28 - Função: definição
 - <https://youtu.be/JiltwM2iqJE>
 - Aula 29 - Função: declaração
 - <https://youtu.be/yhC4MAqHBPw>
 - Aula 30 - Função: variáveis e escopo
 - <https://youtu.be/eglV7E7Kvdl>

Material Complementar

- Vídeo Aulas
 - Aula 31 - Função: passagem de parâmetros
 - <https://youtu.be/JFpodcSZsHw>
 - Aula 32 - Função: recursão (parte 1)
 - https://youtu.be/O_z52PzkfY
 - Aula 33 - Função: recursão (parte 2)
 - <https://youtu.be/cIIhHcHB9n8>