Recursividade
ou
Recursão

Prof. Fabio Fernando Kobs, Dr.

Recursividade ou Recursão

Recursão é uma técnica de programação que permite uma função chamar a si mesma.

Para resolver um problema, a função divide em dois casos: **base** e **geral**.

Algum valor de seus argumentos fará com que uma função recursiva **retorne sem chamar a si mesmo (caso base)**.

Quando a instância mais interna de uma função recursiva retorna, o processo “se desenrolará” completando instâncias pendentes da função, indo da última de volta para a chamada original.

Recursividade - Operacionalização

- Uma **função recursiva é chamada** para resolver um problema.
- A função é **capaz de resolver o caso mais simples do problema** (caso base), ou seja, quando a função é chamada com um caso base, um resultado é retornado.
- Se a função for invocada para um **problema mais complexo**, ela o divide em duas partes conceituais: **uma que a função sabe resolver (base) e uma que não sabe (geral)**.
- Para que a recursão funcione, a parte que ela não sabe resolver **deve se parecer com o problema original**.
- A função então **chama a si própria** para tentar resolver o problema menor (chamada recursiva).
- A função é executada novamente, sobre uma versão mais simples do problema **até alcançar o caso-base**.
- Como a função sabe resolver o caso-base, **seu resultado é retornado à função chamadora, que então consegue resolver, e retorna o resultado à sua chamadora, e assim por diante**.
- A função recursiva **termina quando a chamada original da função receber um retorno**.

Recursividade

Em termos gerais, **toda vez que uma função é iniciada recursivamente, um novo conjunto de variáveis locais e de parâmetros é alocado, e somente esse novo conjunto pode ser referenciado dentro dessa chamada**.

Quando ocorre um **retorno da função para um ponto numa chamada anterior, a alocação mais recente dessas variáveis é liberada, e a cópia anterior é reativada**. Essa cópia anterior é a alocada durante a entrada inicial para a chamada anterior, e é local para essa chamada.

Recursividade - Tipos

Recursão linear: um método é definido a fazer, no máximo, uma chamada recursiva de cada vez que é ativado. Exemplo:

```
return fat(n-1) * n
```

Recursão binária: quando um algoritmo faz duas chamadas recursivas. Exemplo:

```
return SomaBi(a,i,n/2) + SomaBi(a,i + n/2,n/2)
```

Recursão múltipla: generalizando a partir da recursão binária, usa-se recursão múltipla quando um método pode fazer várias chamadas recursivas, em um número maior que dois.
Exemplo: quebra-cabeças com base de 3 pinos (Torres de Hanói).

Recursividade ou Recursão

Presume-se então, que o método que está sendo computado, já foi escrito, e o utiliza em sua própria definição (ex: fat(4) = 4 * fat(3)).

Objetivo fundamental de uma rotina recursiva: conseguir definir um caso “complexo” em termos de um caso “mais simples” (geral) e ter um caso “trivial” (não-recursivo (base)) diretamente solucionável.

Recursividade – Exemplo 1

Cálculo do fatorial:

n! = n * (n-1)! * (n-2)! * ... * 1

Sendo que 0! e 1! são iguais a 1.

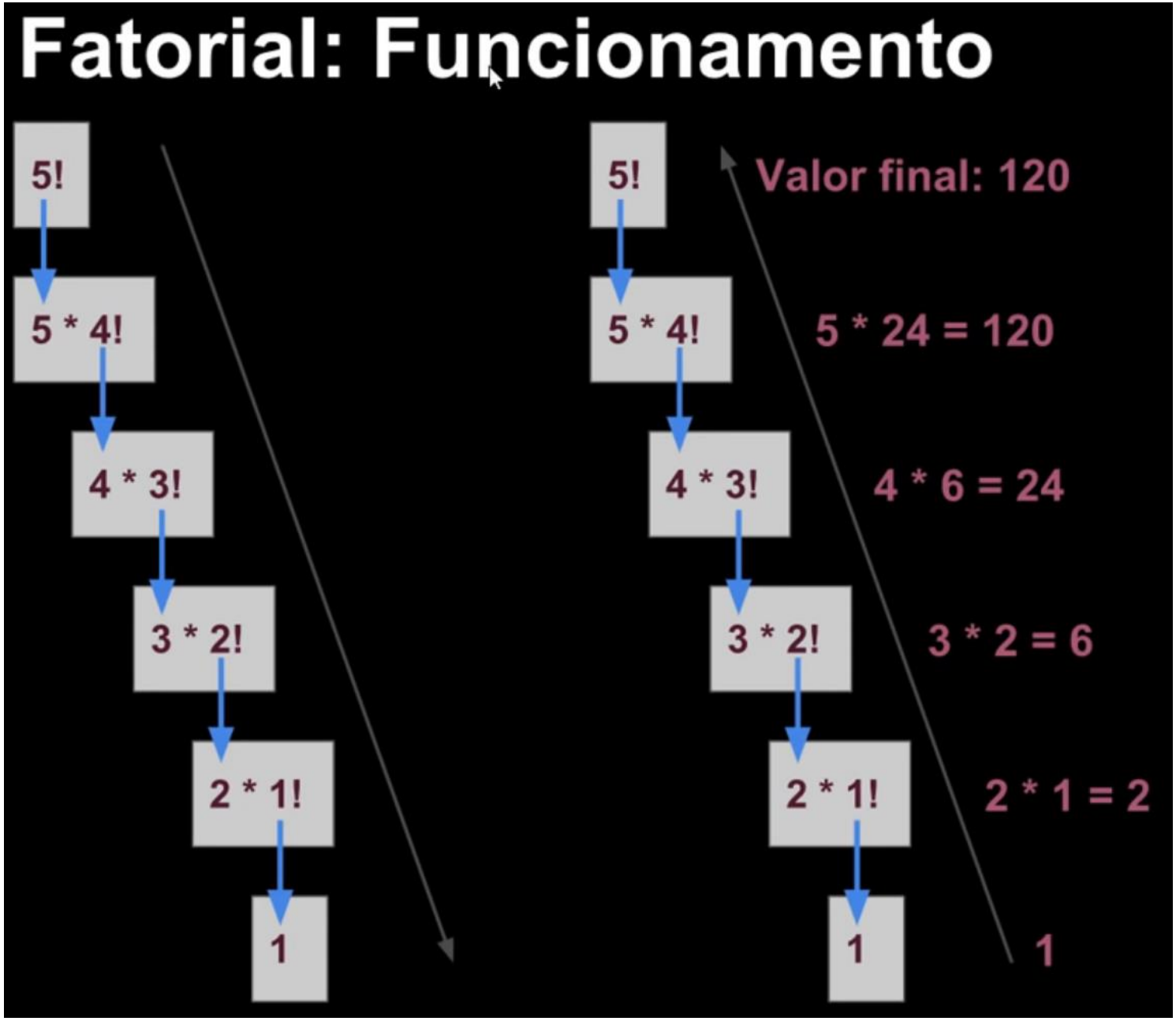
Assim, 5! é igual a: 5! = 5 * 4 * 3 * 2 * 1 = 120

Percebe-se uma sequência de operações semelhantes, multiplicações cada vez mais simples até chegar em 1. Pode-se então resolver utilizando recursão!!!

De forma geral:

$$n! = \begin{cases} \textbf{1 se n=0 ou 1 (caso base)} \\ \textbf{n * (n-1)! se n > 1 (caso geral)} \end{cases}$$

Recursividade – Exemplo 1



Recursividade – Exemplo 1

Cálculo recursivo do fatorial:

```
def fatorial(n):
    if n > 0:
        return n * fatorial(n-1)
    else:
        return 1


numero = int(input('Calcular o fatorial de: '))
print('O fatorial de', numero, 'é', fatorial(numero))
```

Recursividade

Vamos simular a abordagem recursiva
em uma abordagem baseada em pilhas?

Recursividade

Essa descrição sugere o **uso de uma pilha** (Figura 1) **para manter as sucessivas gerações de variáveis locais e parâmetros**.

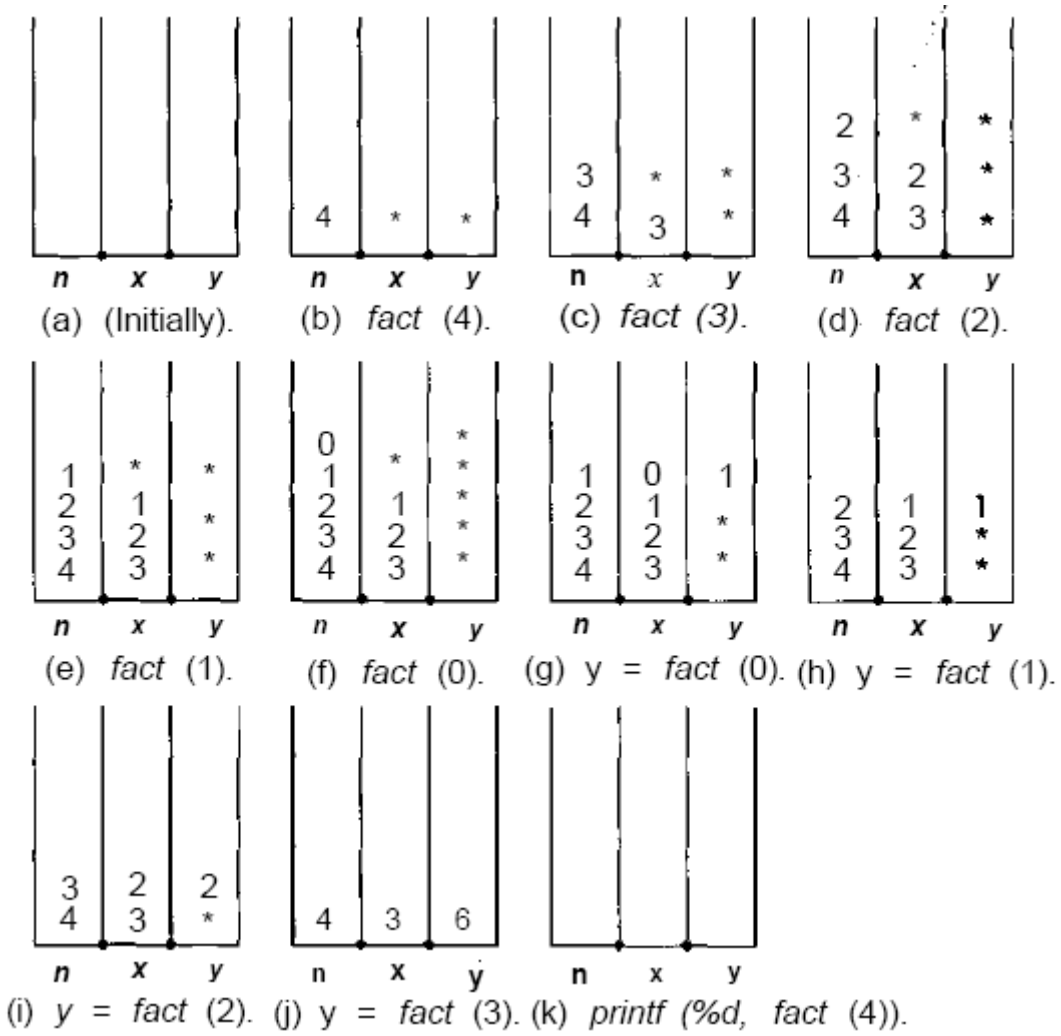
Essa pilha é mantida pelo sistema e é invisível para o usuário.

Toda vez que um método recursivo é iniciado (*push*), uma nova alocação de suas variáveis é introduzida no topo da pilha.

Qualquer referência a uma variável local ou a um parâmetro é feita por meio do topo da pilha atual.

Quando o método retorna, a pilha é esvaziada (*pop*), a alocação do topo é liberada e a alocação anterior torna-se o atual topo da pilha a ser usado para referenciar variáveis locais.

Recursividade



Observe que, toda vez que um método recursivo retorna, ele volta ao ponto imediatamente seguinte ao ponto a partir do qual foi chamado.

Sendo assim, a chamada recursiva a fact(3) retorna à atribuição do resultado a y dentro de fact(4), mas a chamada recursiva a fact(4) retorna ao comando printf no método de chamada.

Figura 1: A pilha de recursão para o cálculo do fatorial, em vários instantes, durante a execução (asterisco significa um valor não inicializado).
Fonte: TENENBAUM (1995, p. 147).

Recursividade – Exemplo 2

Simulação do cálculo recursivo de fatorial, empregando a estrutura de Pilha encadeada:

```
class Node:
    def __init__(self):
        self.__n = 0
        self.__x = 0
        self.__y = 0
        self.__proximo = None
    def getN(self):
        return self.__n
    def setN(self, n):
        self.__n = n
    def getX(self):
        return self.__x
    def setX(self, x):
        self.__x = x
    def getY(self):
        return self.__y
    def setY(self, y):
        self.__y = y
    def getProximo(self):
        return self.__proximo
    def setProximo(self, proximo):
        self.__proximo = proximo
```

Recursividade – Exemplo 2

```
# main

p = Pilha()
numero = int(input('Calcular o fatorial de: '))
p.push(numero)
p.pop()
```

Recursividade – Exemplo 2

```
class Pilha:

    def __init__(self):
        self.__topo = None

    def getTopo(self):
        return self.__topo

    def setTopo(self, topo):
        self.__topo = topo
```

```
def push(self, num):
        temp_no = Node()
        if temp_no:
            if num > 0:
                temp_no.setN(num)
                temp_no.setX(num-1)
                temp_no.setProximo(self.__topo)
                self.__topo = temp_no
                self.printall(1)
                self.push(num-1)
            else:
                temp_no.setN(0)
                temp_no.setX(0)
                temp_no.setY(1)
                temp_no.setProximo(self.__topo)
                self.__topo = temp_no
                self.printall(1)
```

Recursividade – Exemplo 2

```
def pop(self):
        if self.__topo:
            temp_no = self.__topo
            self.setTopo(self.__topo.getProximo())
            if self.__topo:
                if self.__topo.getX() != 0:
                    self.__topo.setY(self.__topo.getX() * temp_no.getY())
                else:
                    self.__topo.setY(1)
                self.printall(2)
                self.pop()
            else:
                if temp_no.getN() * temp_no.getY() != 0:
                    print("Fatorial:", temp_no.getN() * temp_no.getY())
                else:
                    print("Fatorial: 1")
                self.printall(2)
        else:
            print("Pilha vazia...")
```

Recursividade – Exemplo 2

```
def printall(self, push_pop):
    if not self.__topo:
        print("Pilha vazia...")
        return
    temp_no = self.__topo
    saida = "Pilha:  n  x  y\n"
    while temp_no:
        if push_pop == 1:
            saida += "Push    "
        else:
            saida += "Pop     "
        saida += str(temp_no.getN()) + "  " + str(temp_no.getX()) + \
                 "  " + str(temp_no.getY()) + "\n"
        temp_no = temp_no.getProximo()
    print(saida)
```

Recursividade – Torres de Hanói

- O problema consiste em passar todos os discos de um pino para outro qualquer, usando um dos pinos como auxiliar, de maneira que um disco maior nunca fique em cima de outro menor em nenhuma situação.
- O número de discos pode variar, sendo que o mais simples contém apenas três.

Recursividade – Torres de Hanói

O objetivo é deslocar os discos para a terceira estaca (C), usando a segunda estaca (B) como auxiliar.

Somente o primeiro disco de toda estaca pode ser deslocado para outra estaca, e um disco maior não pode nunca ficar posicionado sobre um disco menor. Procure descobrir uma solução.

Recursividade – Torres de Hanói

Solução Recursiva

- A solução para o problema da Torre de Hanoi com recursividade é compacta e baseia-se no seguinte:
  - A única operação possível de ser executada é "move disco de um pino para outro";
  - Uma torre com (N) discos, em um pino, pode ser reduzido ao disco de baixo e a torre de cima com (N-1) discos;
  - A solução consiste em transferir a torre com (N-1) discos do pino origem para o pino auxiliar, mover o disco de baixo do pino origem para o pino destino e transferir a torre com (N-1) discos do pino auxiliar para o pino destino. Como a transferência da torre de cima não é uma operação possível de ser executada, ela deverá ser reduzida sucessivamente até transformar-se em um movimento de disco.

Recursividade – Torres de Hanói

Solução Ótima

- O número mínimo de "movimentos" para conseguir transferir todos os discos é 2^n-1, sendo **n** o número de discos.
- Logo:
  - Para 3 discos , são necessários 7 movimentos
  - Para 7 discos, são necessários 127 movimentos
  - Para 15 discos, são necessários 32.767 movimentos
  - Para 64 discos, como diz a lenda, são necessários 18.446.744.073.709.551.615 movimentos.

Recursividade - Vantagens

- É possível identificar um número considerável de variáveis locais e temporárias que não precisam ser salvas e restauradas pelo uso de uma pilha;

- Às vezes, uma solução recursiva é o método mais natural e lógico de solucionar um problema, por exemplo, não é certo que um programador consiga desenvolver a solução não-recursiva para o problema das Torres de Hanói diretamente a partir da declaração do problema;

- Clareza na leitura do código, simplicidade;

Recursividade - Vantagens

• Quando uma pilha não pode ser eliminada da versão não-recursiva de um programa e quando a versão recursiva não contém nenhum dos parâmetros adicionais ou variáveis locais, a versão recursiva pode ser tão ou mais veloz que a versão não-recursiva, sob um compilador eficiente. As Torres de Hanoi representam um exemplo desse programa recursivo;

• Outro exemplo de recursividade eficiente (no percurso de árvores ordenadas);

• As ideias e transformações da função fatorial e o problema das Torres de Hanoi podem ser aplicadas a problemas mais complexos, cuja solução não-recursiva não esteja prontamente evidente.

Recursividade - Desvantagens

• Em geral, uma versão não-recursiva de um programa executará com mais eficiência, em termos de tempo e espaço, do que uma versão recursiva. Isso acontece porque o trabalho extra despendido para entrar e sair de um bloco é evitado na versão não-recursiva;

• O fatorial, cuja versão não-recursiva não precisa de uma pilha, e o cálculo de números de Fibonacci (não precisa de uma pilha também), é um exemplo onde a recursividade pode ser evitada numa implementação prática.

Recursividade – Exercícios

01) Construa um programa para calcular a soma sucessiva de um inteiro positivo qualquer, mediante uma função recursiva.

02) Sabe-se que a sequência de Fibonacci é a sequência de inteiros:
0 , 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , 34 . . . Se permitirmos que fib(0) = 0, fib(l) = 1, fib(2) = 1, fib(6) = 8, e assim por diante, então construa uma função recursiva para determinar o valor Fibonacci.

03) Faça um programa para a multiplicação de números naturais. O produto a * b, em que a e b são inteiros positivos, pode ser definido como a somado a si mesmo b vezes. Essa é uma definição iterativa. Por exemplo, para avaliar 6*3, avalia-se primeiro 6*2 e depois soma-se 6. Para avaliar 6*2, avalia-se primeiro 6*1 e soma-se 6. Mas 6*1 é igual a 6. Sendo assim: 6*3=6*2+6=6*1+6+6=6+6+6=18.

Recursividade – Exercícios

04) Implemente o exercício 01 mediante o uso de pilhas encadeadas.

05) Implemente o exercício 02 mediante o uso de pilhas encadeadas.

06) Implemente o exercício 03 mediante o uso de pilhas encadeadas.

07) A sequência S é definida recursivamente por:

1. S(1) = 2
2. S(2) = 2S(n-1) para n >= 2

Pela proposição 1, S(1), o primeiro objeto em *S*, é 2. Então pela proposição 2, o segundo objeto em *S* é S(2) = 2S(1) = 2(2) = 4.
Como entrada tem-se um inteiro positivo e maior ou igual a 1. A saída é outro inteiro positivo. Assim:
a) Escreva um programa recursivo para determinar a sequência S(n).
b) Usando o programa desenvolvido, elabore o teste de mesa a fim de determinar o resultado para S(5).

Recursividade – Exercícios

08) A função de Ackermann (por Wilhelm Ackermann, matemático alemão), que cresce mais rapidamente do que uma função exponencial (e até mais do que uma função exponencial múltipla), tem a seguinte definição, sendo m e n inteiros não-negativos:

```
             n + 1                      se m = 0
A(m, n) =    A(m - 1, 1)                se m > 0 e n = 0
             A(m - 1, A(m, n - 1))      se m > 0 e n > 0
```

Dois inteiros positivos (m, n) são a entrada. A saída é outro inteiro positivo. Assim:
a) Escreva um programa recursivo para calcular A(m, n).
b) Usando a definição da função de Ackermann, determine e demonstre o resultado de A(2,2).

09 - Bônus) Construa um programa recursivo para resolver o problema das Torres de Hanói.

Recursividade – Exercícios – Definições

01) Soma sucessiva
Definição:
0 se n == 0
n + soma(n-1) se n >= 1

02) Fibonacci
Definição:
fib(n) = n if n == 0 or n == 1
fib(n) = fib(n - 2) + fib(n - 1) if n >= 2

03) Número natural: Um número natural é um número inteiro não-negativo (0, 1, 2, 3, ...). Em alguns contextos, número natural é definido como um número inteiro positivo, i.e., o zero não é considerado como um número natural.
Definição:
a * b = a se b == 1
a * b = a + mult(a, b - 1) se b > 1