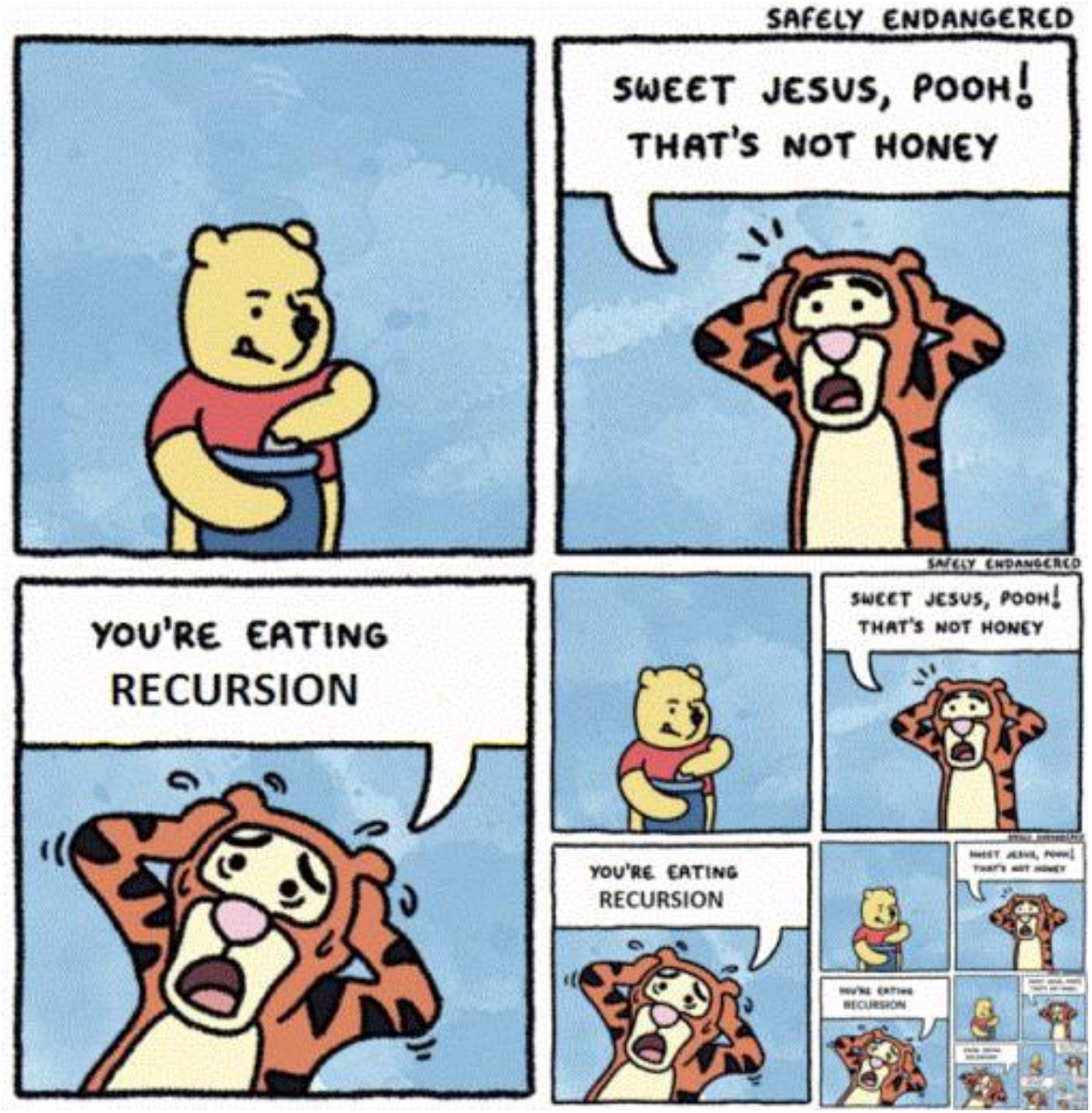
Recursividade

Fabio Lubacheski
fabio.lubacheski@mackenzie.br

Recursividade

Recursividade



Introdução

Considere a recorrência do fatorial:

$$n! = \begin{cases} 1 & se\ n = 0 \\ n\,.(n-1)!\ & se\ n > 0 \end{cases}$$

A implementação de uma função que calcula fatorial com os conhecimentos que temos até agora poderia ser descrita assim:

```
def fatorial( n ):
    fat = 1
    while termo <= n:
        fat = fat*termo
        termo = termo + 1

    return fat
```

Recursividade

A recursividade permite solucionar um problema a partir de uma instância do problema conhecida, e as soluções para as outras instâncias são conhecidas a partir desta.

Um problema é dito **recursivo** se ele for definido em **termos de si próprio**, ou seja, recursivamente. Por exemplo a definição do fatorial.

$$n! = \begin{cases} 1 & se\ n = 0 \\ n.(n-1)!\ & se\ n > 0 \end{cases}$$

Recursividade

Dado que o problema é recursivo podemos escrever um algoritmo recursivo com a seguinte estratégia:

se a entrada do problema é pequena e conhecida então

resolva-a diretamente e **pare a recursão**;

senão,

reduza-a uma entrada menor do mesmo problema,

aplique este método à entrada menor

e volte à entrada original.

Recursividade

A estrutura recursiva do cálculo do fatorial fica evidente. Assim, teremos a seguinte implementação recursiva.

```
def fatorial( n ):
    #base da recursão (condição de parada)
    if n == 0:
        return 1
    else:
        return n * fatorial(n-1)
```

Uma implementação recursiva de uma se caracteriza por conter no corpo da função uma chamada para si mesma.

A chamada de si mesma é dita **chamada recursiva**.

Recursividade

Outros exemplos:

https://www.cs.usfca.edu/~galles/visualization/RecFact.html

https://www.cs.usfca.edu/~galles/visualization/RecReverse.html

https://www.cs.usfca.edu/~galles/visualization/RecQueens.html

https://www.youtube.com/watch?v=hLnuMXO95f8

Como testar e entender um algoritmo recursivo ?

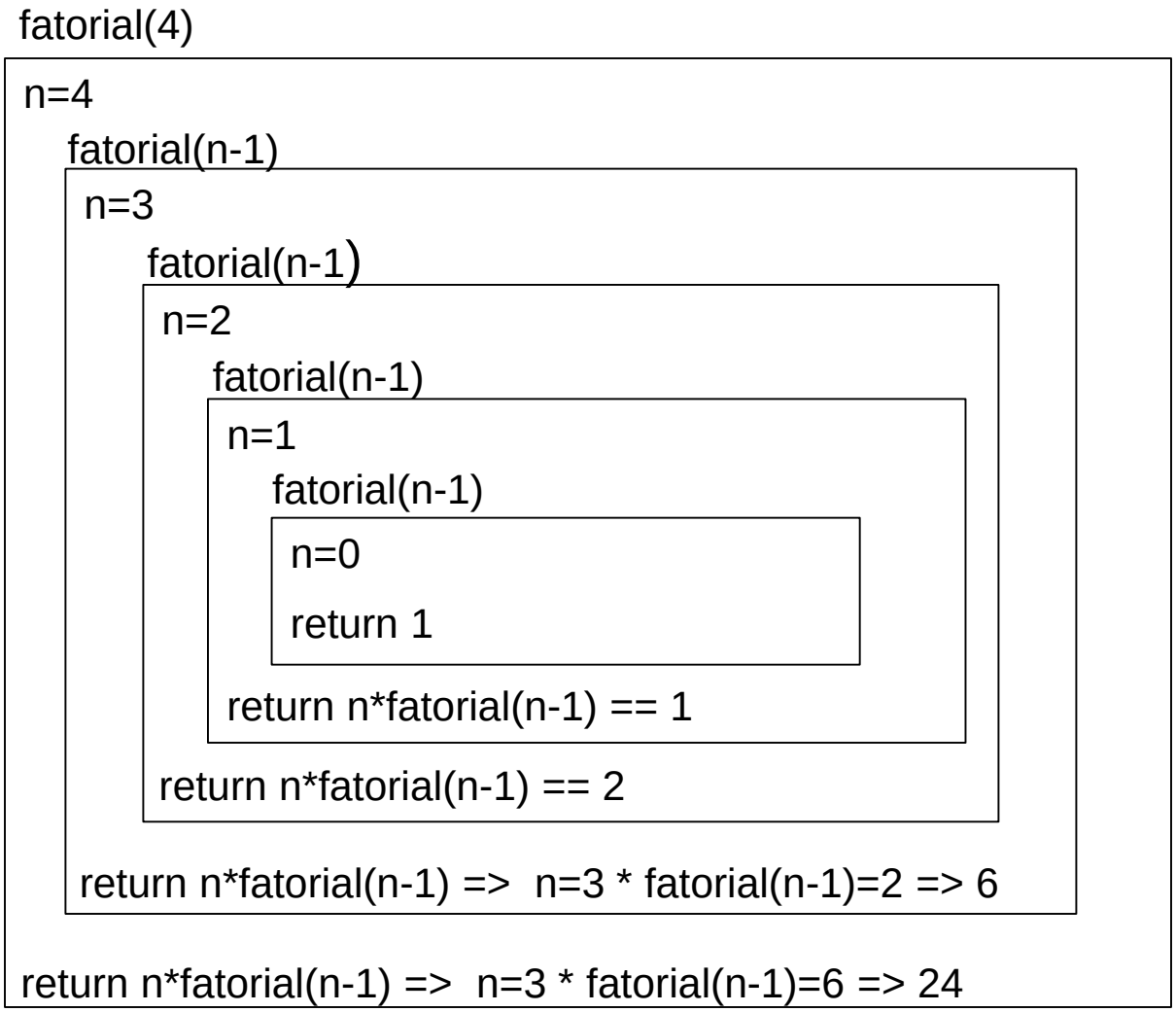
Recursividade - Entendimento

O **diagrama de execução** é um processo manual que é utilizado para validar e simular a lógica de uma **função recursiva**.

- Cada chamada recursiva da função é representada por um retângulo;
- Uma nova chamada recursiva resulta em um novo retângulo que estará encaixado no retângulo anterior, e assim sucessivamente.
- Na parte superior de cada retângulo são representadas as variáveis locais e parâmetros da função.
- Na parte inferior é representado o retorno da função
- No diagrama de execução não são representadas as chamadas de outras funções iterativas.
- Convenções adicionais serão indicadas nos exemplos que se seguem.

Diagrama de execução

Diagrama de execução para fatorial(4):



Recursivo x Iterativo

Se compararmos as implementações iterativa e recursiva do fatorial, não é difícil nos convencermos de que a **recursiva** é **mais simples**.

Embora a implementação seja simples, o custo de uma execução recursiva pode ser alto. Cada chamada recursiva irá empilhar dados na pilha de execução do programa (Stack) e, caso esta pilha não dimensionada corretamente, podemos ter um “estouro de pilha”. Além disto, o processo de empilhamento pode ter um impacto significativo no desempenho do programa.

Recursivo x Iterativo

OBSERVAÇÕES IMPORTANTES:

1. Todo algoritmo **recursivo** tem uma versão **iterativa**, mas nem todo algoritmo **iterativo** tem uma versão **recursiva**
2. A versão iterativa de uma algoritmo é sempre mais rápida que sua versão recursiva.
3. A complexidade de um algoritmo iterativo é a mesma que sua versão recursiva.

Sequência de Fibonacci

A sequência [0, 1, 1, 2, 3, 5, 8, 13, 21, ...] é conhecida como sequência ou série de Fibonacci, e tem aplicações teóricas e práticas, na medida em que alguns padrões na natureza parecem segui-la. Pode ser obtida através da recorrência abaixo:

$$\mathrm{Fib}(n) = \begin{cases} 0 & se\ n = 0 \\ 1 & se\ n = 1 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & se\ n > 1 \end{cases}$$

Como ficaria a implementação do algoritmo ?

Como ficaria o diagrama de execução para Fib(4) ?

Busca binária

Considere o algoritmo de busca binária iterativa:

```
def BuscaBinaria( v, x ):
      i = 0 #inicio do vetor
      f = len(v)-1 #fim do vetor
      while i <= f:
            m = (i + f)//2 #divisao inteira
            if v[m] == x:
                  return m

            if v[m] < x:
                  i = m + 1
            else:
                  f = m - 1
      return -1
```

Busca binária recursiva

Para implementarmos a busca binária recursiva será necessário *generalizar* ligeiramente o problema, trocando `v[0..n-1]` por `v[i..f]`. Assim teríamos que a função recebe um número `x` e um vetor em ordem crescente `v[i..f]`. Ele devolve um índice m tal que `v[m] == x` ou devolve `-1` se tal `m` não existe. A declaração da função ficaria:

```
def BuscaBinaria( v, x, i, f ):
```

Exercícios

1) Considere a seguinte função abaixo:

```
def result( n ):
  if n == 1:
     return 2
  else:
     return 2 * result(n - 1)
```

Faça o diagrama de execução para **`result(5)`**

2) Faça o diagrama de execução para **`recursao(27)`**

```
def recursao( n ):
   if n <= 10:
      return n * 2
   else:
      return recursao(recursao(n/3))
```

Exercícios

3) Qual a "profundidade da recursão" da função que realiza a busca binária recursiva ? Ou seja, quantas vezes `busca()` chama a si mesma?

4) Implemente uma função recursiva para calcular a potência $\mathbf{a^n}$, supondo que tanto **a** quanto **n** sejam números inteiros positivos.

5) Dada um vetor de números inteiros positivos, descreva funções recursivas para :

a) Fazer a busca linear de um elemento no vetor;

b) Encontrar o menor elemento no vetor;

c) Fazer a soma dos elementos no vetor;

c) Calcular a média aritmética dos elementos no vetor;

Exercícios

6) Faça o diagrama de execução para a função recursiva abaixo com `N=7`.

```
def fusc( n ):

    if  n <= 1:
      return 1

    if n % 2 == 0:
      return fusc( n/2 )

    return fusc((n-1)/2 ) + fusc((n+1)/2 )
```

Exercícios

7) A função abaixo calcula o máximo divisor comum dos inteiros positivos m e n. Escreva uma função recursiva.

```
def mdc( m, n ):

    while n != 0:
        r = m % n
        m = n
        n = r

    return m
```

Exercícios

8) Escreva uma função recursiva que calcula o produto de $a * b$, em que a e b são inteiros maiores que zero. considere que o produto pode ser definido como a somado a si mesmo b vezes, usando uma definição recursiva temos

$a * b = a$ se $b = 1$

$a * b = a * (b - 1) + a$ se $b > 1$

9)Escreva uma função recursiva (ou iterativa) que gera todos as permutações (combinação simples sem repetição de valores) dígitos 1, 2, 3 .. n armazenados no vetor A[0.. n-1]. Para de n=3 teríamos os seguintes dígitos 1, 2 e 3 armazenados no vetor A[] e como saída 6 permutações.

1= 1 2 3
2= 1 3 2
3= 2 1 3
4= 2 3 1
5= 3 1 2
6= 3 2 1

Como dica use um vetor auxiliar para gerar as permutações.

Exercícios

11) A multiplicação Russa consiste em:

a. Escrever os números A e B, que se deseja multiplicar na parte superior das colunas.

b. Dividir A por 2, sucessivamente, ignorando o resto até chegar à unidade, escrever os resultados da coluna A.

c. Multiplicar B por 2 tantas vezes quantas se haja dividido A por 2, escrever os resultados sucessivos na coluna B.

d. Somar todos os números da coluna B que estejam ao lado de um número ímpar da coluna A.

Exemplo: 27 × 82 = **2214**

```
A   B    Parcelas
27  82   82
13  164  164
6   328  –
3   656  656
1   1312 1312
```

Soma: 2214

Programar em Python uma função recursiva que permita fazer à multiplicação russa de 2 entradas;

Fim