

Introdução à Programação

Aula 16: Funções (parte 2): Recursividade

Prof. Eduardo Corrêa Gonçalves

06/06/2024

Sumário

Introdução

O que é Recursividade?

Ativação de Funções

Recursividade em Python

Anatomia de uma Função Recursiva

Recursividade em Ação – “Potência de 2”

Outros Exemplos

Considerações Adicionais

Recursividade Mútua

Solução Recursiva x Solução Iterativa

Introdução (1/6)

• Recursividade

- Uma função é dita **recursiva** quando possui em seu corpo uma **chamada para ela própria**.
- A função “`contagem_regressiva`” ao lado é um exemplo.
- Essa aula, aborda a recursividade em Python.
 - Mas antes de começar a falar do assunto, vamos apresentar o conceito de **pilha de ativação**.

```

1 def contagem_regressiva(n):
2     if (n >= 1):
3         print(n)
4         contagem_regressiva(n-1)
5     else:
6         print('fogoooo!!!')
7
8
9 k=3
10 contagem_regressiva(k)
--

```

Shell ×

```

>>> %Run contagem_regressiva.py

3
2
1
fogoooo!!!

```

Introdução (2/6)

- **Pilha de Ativação (1/5)**

- Sempre que um programa é executado, **duas áreas** principais de memória são organizadas:
 - **Área do programa principal**: para manter as variáveis do programa principal.
 - **Pilha de ativação**: área organizada em forma de pilha para armazenar os **parâmetros** e **variáveis locais** de uma **função em execução**.
- Melhor explicando:
 - Sempre que uma função é **ativada** (chamada), um espaço contendo os valores dos parâmetros e variáveis locais (**contexto da chamada**) é alocado no topo da pilha.
 - Ao fim da função, o espaço é automaticamente desalocado.

Introdução (3/6)

- Pilha de Ativação (2/5)
 - **Exemplo:** programa que computa o sucessor de um número.

```
#função "sucessor"
def sucessor(numero):
    return numero + 1

#programa principal
a=100
b=sucessor(a)
print('b =', b)
```

- **Obs.:** a função “sucessor” **não** é recursiva!
 - Porém, a pilha de ativação é criada para qualquer tipo de função, seja ela recursiva ou não.

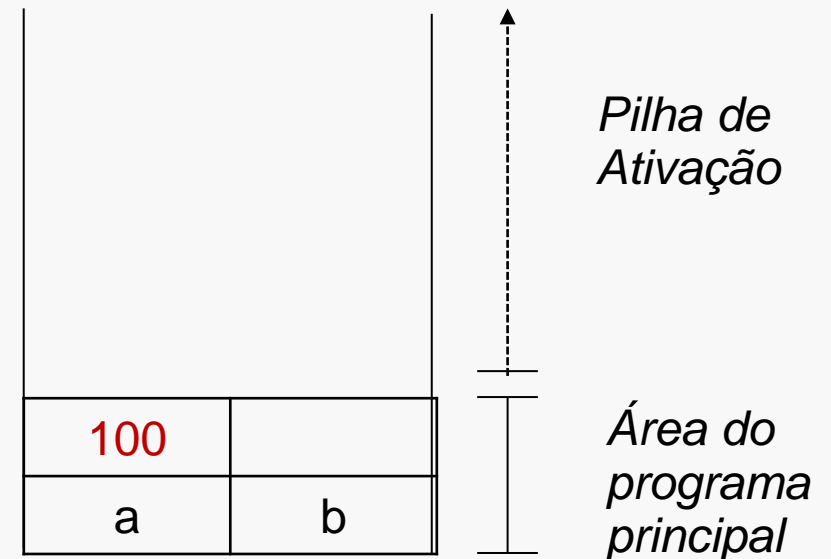
Introdução (4/6)

• Pilha de Ativação (3/5)

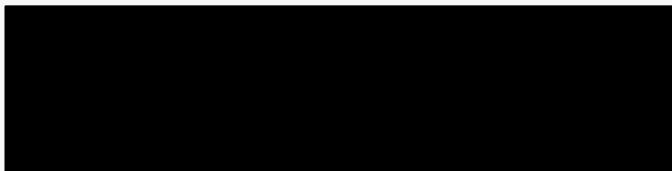
- Ao iniciar o programa, a área de memória do programa principal é alocada e a pilha de ativação está vazia.

```
#função "sucessor"
def sucessor(numero):
    return numero + 1

#programa principal
a=100
b=sucessor(a)
print('b =', b)
```



saída:



Introdução (5/6)

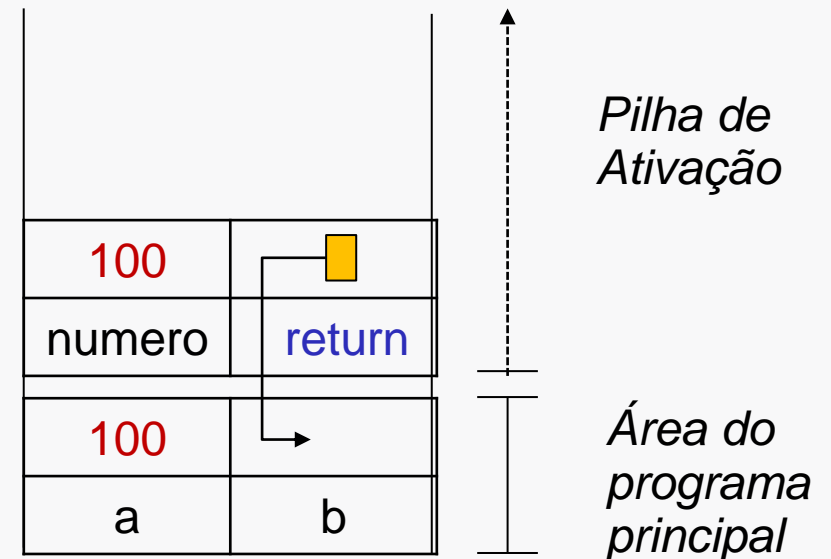
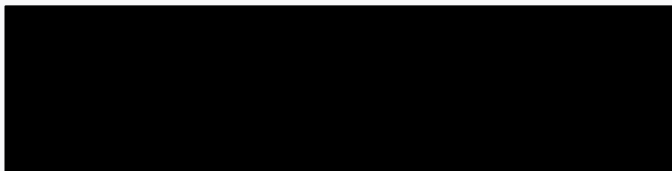
• Pilha de Ativação (4/5)

- No **momento da ativação** da função “sucessor”, um espaço para suas variáveis/parâmetros é alocado no topo da pilha.

```
#função "sucessor"
def sucessor(numero): ← 3
    return numero + 1

#programa principal
a=100
b=sucessor(a) ← 2
print('b =', b)
```

saída:



Introdução (6/6)

- Pilha de Ativação (5/5)
 - Ao fim da execução de “sucessor”, os dados da função são desempilhados.

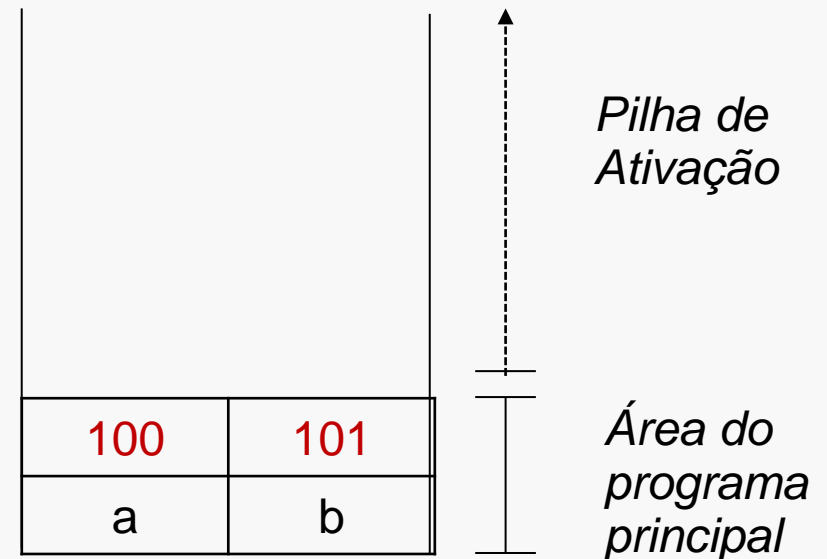
```
#função "sucessor"  
def sucessor(numero):  
    return numero + 1
```

```
#programa principal  
a=100  
b=sucessor(a)  
print('b =', b)
```

4

saída:

b = 101

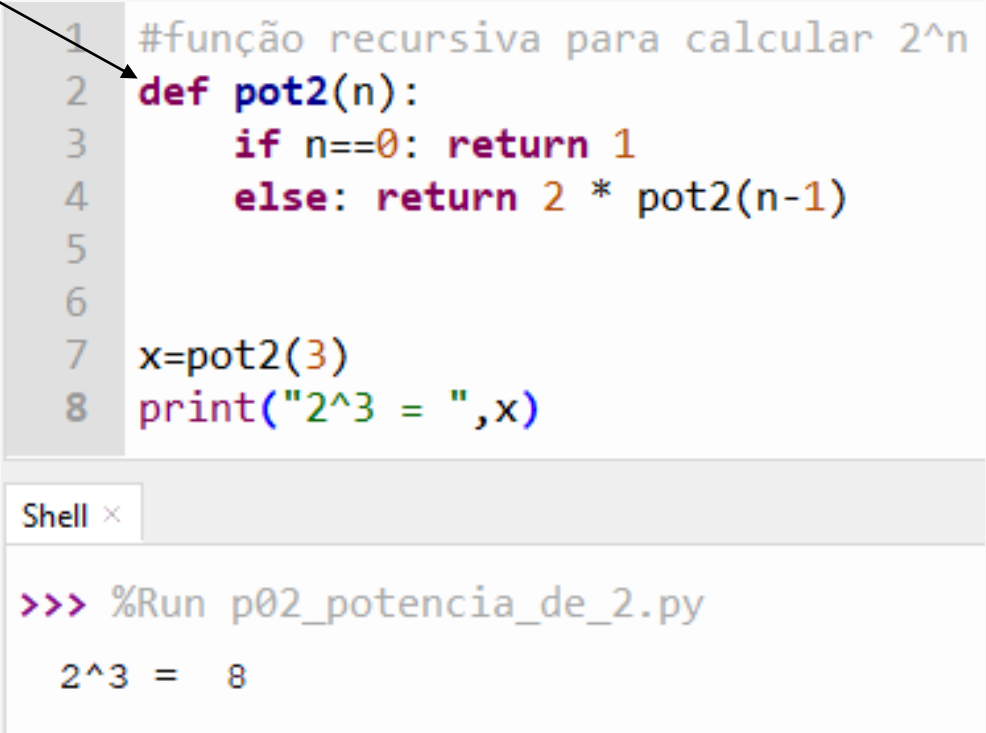


Recursividade em Python (1/19)

- Anatomia de uma Função Recursiva (1/3)

- Uma função que chama a si mesma é dita ser recursiva.
 - A função “**pot2**” é um exemplo. Ela calcula o valor de 2^n .

- A ideia básica de uma função recursiva consiste em **diminuir sucessivamente** o problema original em um subproblema menor.
- O processo executa até que seja possível resolvê-lo de forma direta, sem recorrer a si mesmo.
- Quando isso ocorre, diz-se que a função atingiu uma **condição de parada**.



```
1 #função recursiva para calcular 2^n
2 def pot2(n):
3     if n==0: return 1
4     else: return 2 * pot2(n-1)
5
6
7 x=pot2(3)
8 print("2^3 = ",x)
```

Shell ×

```
>>> %Run p02_potencia_de_2.py
2^3 = 8
```

Recursividade em Python (2/19)

- Anatomia de uma Função Recursiva (2/3)
 - Em **todas** as funções recursivas existe:
 - Uma **chamada recursiva**:
 - Onde a função **chama a si própria**.
 - Normalmente baseada em uma **expressão de recorrência** para resolver um subproblema do problema inicial.
 - Uma **condição de parada**:
 - Também chamada de “caso base” ou “condição de terminação”.
 - Quando atingida, **faz a recursão acabar**.
 - Precisa ser definida, caso contrário o programa entra em loop infinito e estoura a capacidade da pilha de ativação!

Recursividade em Python (3/19)

- Anatomia de uma Função Recursiva (3/3)

$$2^n = \begin{cases} 1, \text{ se } n = 0 & \longrightarrow \text{caso base} \\ 2 \times 2^{n-1}, \text{ se } n > 0 & \longrightarrow \text{expressão de recorrência} \end{cases}$$

```
def pot2(n):  
    if n==0: return 1  
    else: return 2 * pot2(n-1)
```

condição de parada

chamada recursiva

Recursividade em Python (4/19)

- **Recursividade em Ação – Cálculo de 2^n**
 - Será simulada a execução linha por linha do programa abaixo, que contém a função recursiva “`pot2`”.
 - Observaremos o comportamento pilha de ativação durante as chamadas recursivas.

```
#função recursiva para calcular 2^n
def pot2(n):
    if n==0: return 1
    else: return 2 * pot2(n-1)

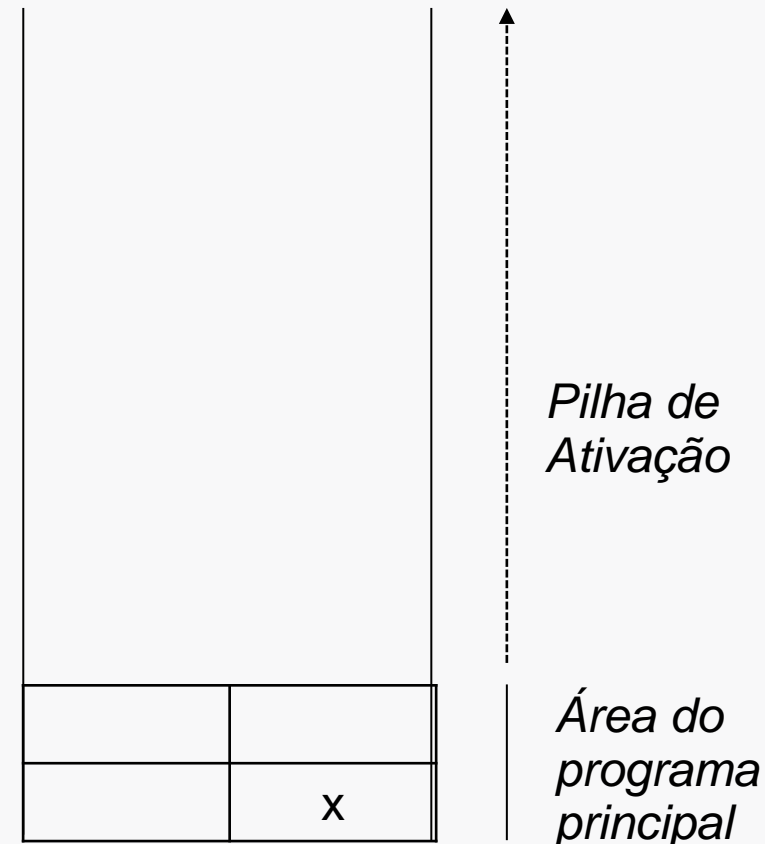
x=pot2(3)
print("2^3 = ",x)
```

Recursividade em Python (5/19)

- **Recursividade em Ação – Cálculo de 2^n**
 - Ao iniciarmos o programa, a área de memória do programa principal é organizada.
 - E a pilha de ativação está vazia.

```
def pot2(n):
    if n==0: return 1
    else: return 2 * pot2(n-1)
```

```
x=pot2(3) ←
print("2^3 = ",x)
```



Recursividade em Python (6/19)

- Recursividade em Ação – Cálculo de 2^n
 - A função “`pot2`” é ativada (chamada) pelo programa principal, com $n=3$.
 - O contexto (dados) da primeira chamada é empilhado.

```
def pot2(n): ←
    if n==0: return 1
    else: return 2 * pot2(n-1)
```

```
x=pot2(3)
print("2^3 = ",x)
```

pot2(3)

3	
n	return
	x

Pilha de Ativação

Área do programa principal

Recursividade em Python (7/19)

- **Recursividade em Ação – Cálculo de 2^n**
 - A condição de parada não foi atingida. Então, realiza-se a primeira chamada recursiva, com $n=2$
 - Ou seja, vai ativar de novo “pot2”, dessa vez com $n=2$.

```
def pot2(n):
    if n==0: return 1
    else: return 2 * pot2(n-1) ←
```

```
x=pot2(3)
print("2^3 = ",x)
```

pot2(3)

3	2*pot(2)
n	return
	x

Pilha de Ativação

Área do programa principal

Recursividade em Python (8/19)

- Recursividade em Ação – Cálculo de 2^n
 - O contexto da nova chamada é empilhado.

```
def pot2(n): ←
    if n==0: return 1
    else: return 2 * pot2(n-1)
```

```
x=pot2(3)
print("2^3 = ",x)
```

pot2(2)

pot2(3)

	2	
	n	return
	3	2*pot(2)
	n	return
		x

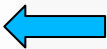
Pilha de Ativação

Área do programa principal

Recursividade em Python (9/19)

- Recursividade em Ação – Cálculo de 2^n
 - A condição de parada ainda não foi atingida, então é feita nova chamada recursiva, com $n=1$.

```
def pot2(n):
    if n==0: return 1
    else: return 2 * pot2(n-1)
```



```
x=pot2(3)
print("2^3 = ",x)
```

pot2(2)

pot2(3)

2	2*pot(1)
n	return
3	2*pot(2)
n	return
	x

Pilha de Ativação

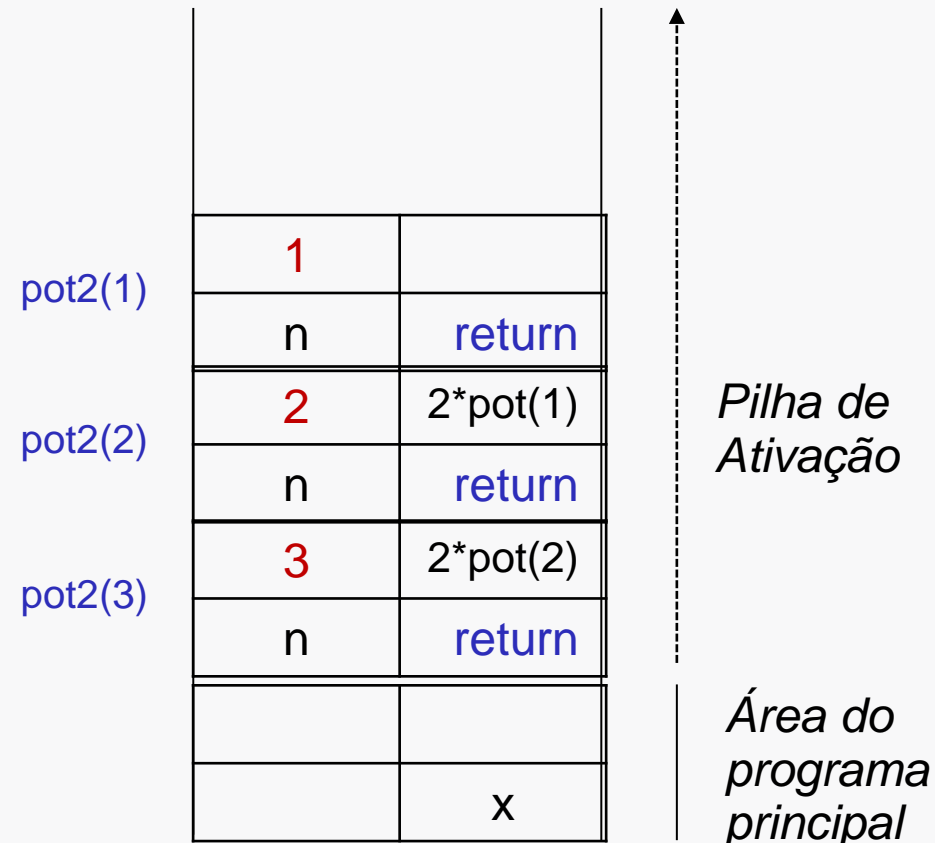
Área do programa principal

Recursividade em Python (10/19)

- Recursividade em Ação – Cálculo de 2^n
 - O contexto da nova chamada é empilhado.

```
def pot2(n): ←
    if n==0: return 1
    else: return 2 * pot2(n-1)
```

```
x=pot2(3)
print("2^3 = ",x)
```



Recursividade em Python (11/19)

- Recursividade em Ação – Cálculo de 2^n
 - A condição de parada ainda não foi atingida, então é feita nova chamada recursiva, com $n=0$.

```
def pot2(n):
    if n==0: return 1
    else: return 2 * pot2(n-1)

x=pot2(3)
print("2^3 = ",x)
```

pot2(1)

pot2(2)

pot2(3)

1	2*pot(0)
n	return
2	2*pot(1)
n	return
3	2*pot(2)
n	return
	x

Pilha de Ativação

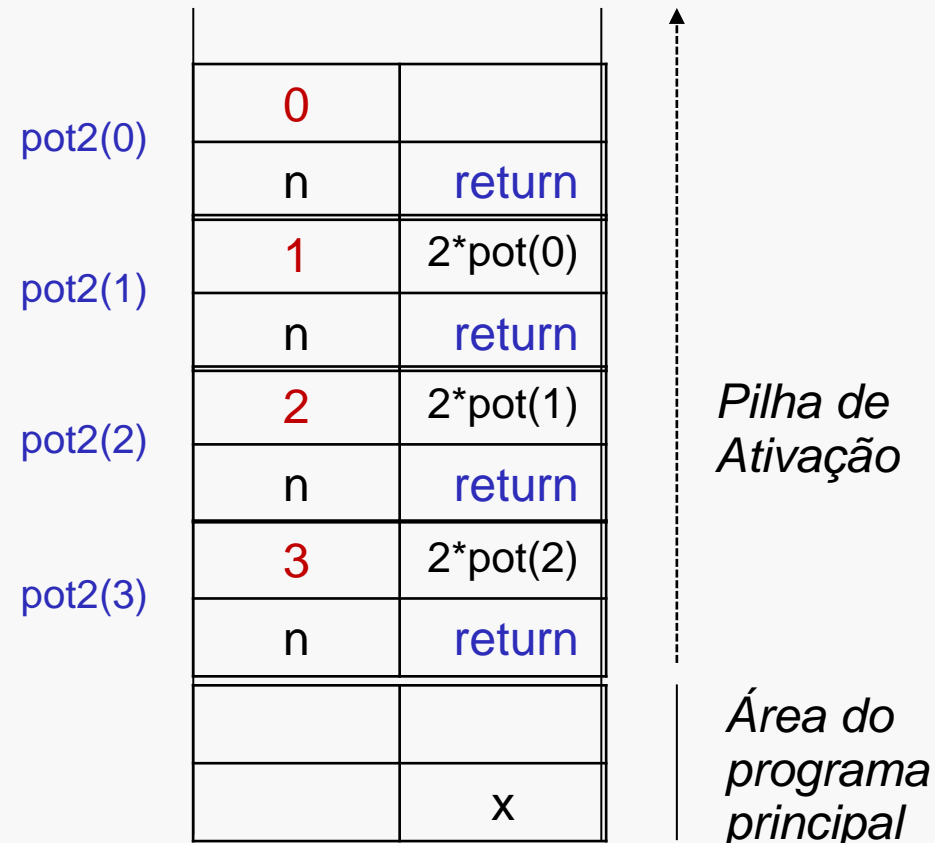
Área do programa principal

Recursividade em Python (12/19)

- Recursividade em Ação – Cálculo de 2^n
 - O contexto da chamada é empilhado...

```
def pot2(n): ←
    if n==0: return 1
    else: return 2 * pot2(n-1)
```

```
x=pot2(3)
print("2^3 = ",x)
```

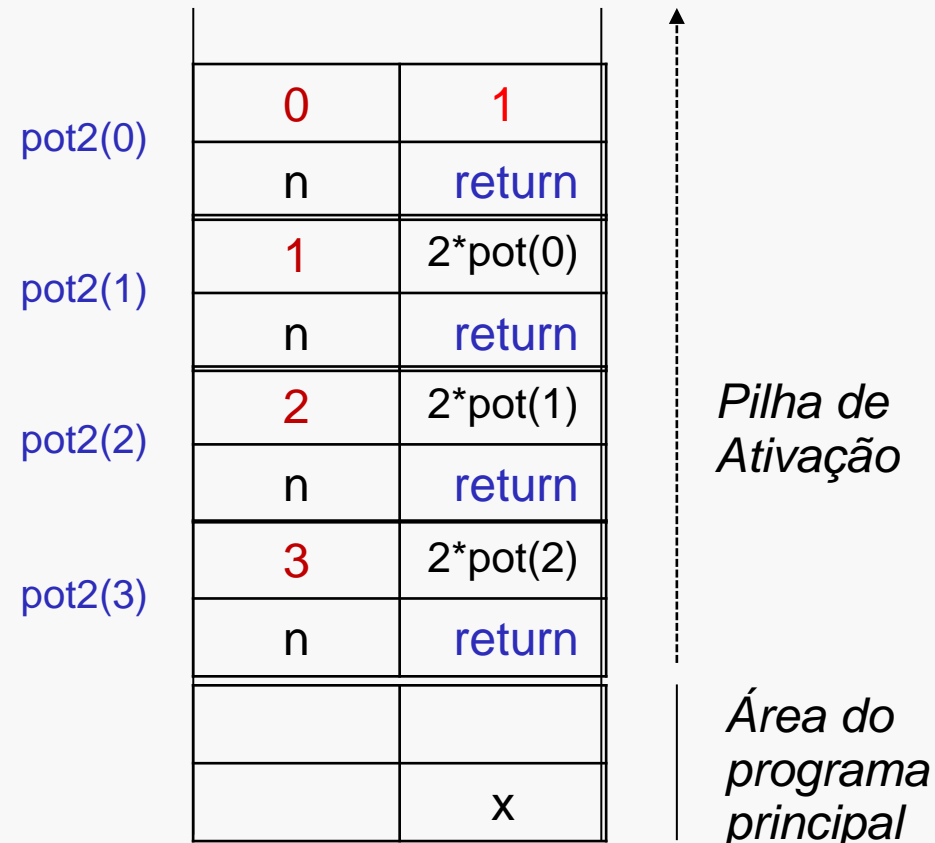


Recursividade em Python (13/19)

- Recursividade em Ação – Cálculo de 2^n
 - Condição de parada atingida!!!
 - O valor 1 será **retornado**;
 - Vai começar a **desempilhar** e **atualizar o cálculo** de 2^n .

```
def pot2(n):
    if n==0: return 1 ←
    else: return 2 * pot2(n-1)
```

```
x=pot2(3)
print("2^3 = ",x)
```



Recursividade em Python (14/19)

- Recursividade em Ação – Cálculo de 2^n
 - Retorna 1, desempilha o contexto de `pot(0)`.

```
def pot2(n):
    if n==0: return 1
    else: return 2 * pot2(n-1)

x=pot2(3)
print("2^3 = ",x)
```

pot2(1)

pot2(2)

pot2(3)

0	1
n	return
1	2*1
n	return
2	2*pot(1)
n	return
3	2*pot(2)
n	return
	x

Pilha de Ativação

Área do programa principal

Recursividade em Python (15/19)

- Recursividade em Ação – Cálculo de 2^n
 - Retorna $2*1=2$, desempilha `pot(1)`.

```
def pot2(n):
    if n==0: return 1
    else: return 2 * pot2(n-1)

x=pot2(3)
print("2^3 = ",x)
```

pot2(2)

pot2(3)

0	1
n	return
1	$2*1=2$
n	return
2	$2 * 2$
N	return
3	$2*pot(2)$
n	return
	x

Pilha de Ativação

Área do programa principal

Recursividade em Python (16/19)

- Recursividade em Ação – Cálculo de 2^n
 - Retorna $2*2=4$, desempilha `pot(2)`.

```
def pot2(n):
    if n==0: return 1
    else: return 2 * pot2(n-1)

x=pot2(3)
print("2^3 = ",x)
```

pot2(3)

0	1
n	return
1	2*1=2
n	return
2	2*2=4
n	return
3	2 * 4
n	return
	x

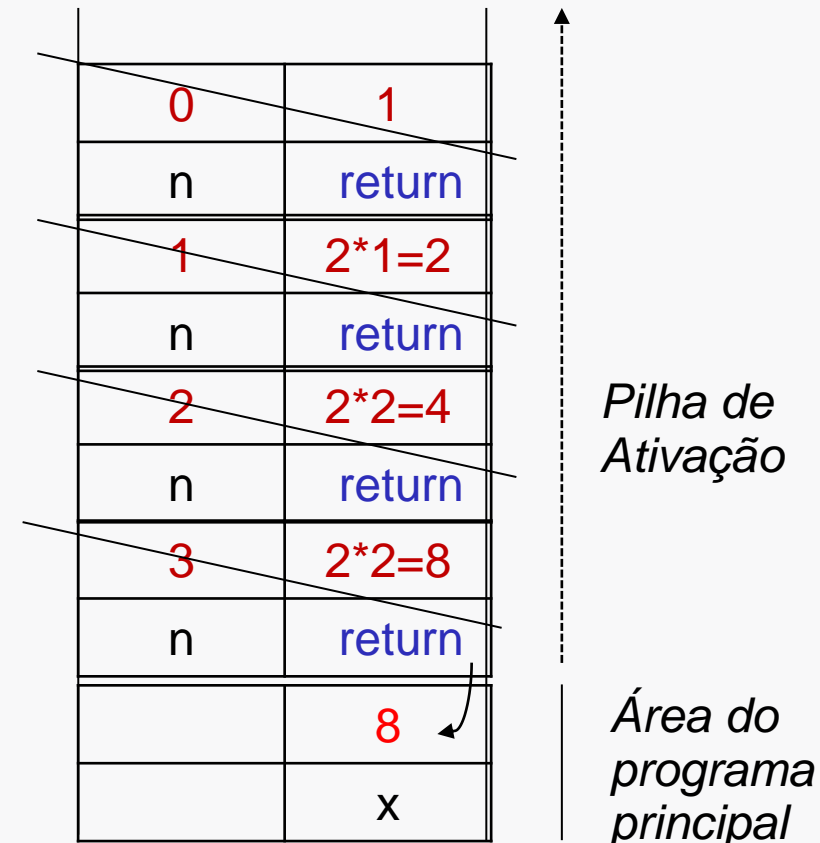
Pilha de Ativação

Área do programa principal

Recursividade em Python (17/19)

- Recursividade em Ação – Cálculo de 2^n
 - Retorna $2*4=8$, desempilha `pot(3)`.

```
def pot2(n):  
    if n==0: return 1  
    else: return 2 * pot2(n-1)  
  
x=pot2(3)  
print("2^3 = ",x)
```



Recursividade em Python (18/19)

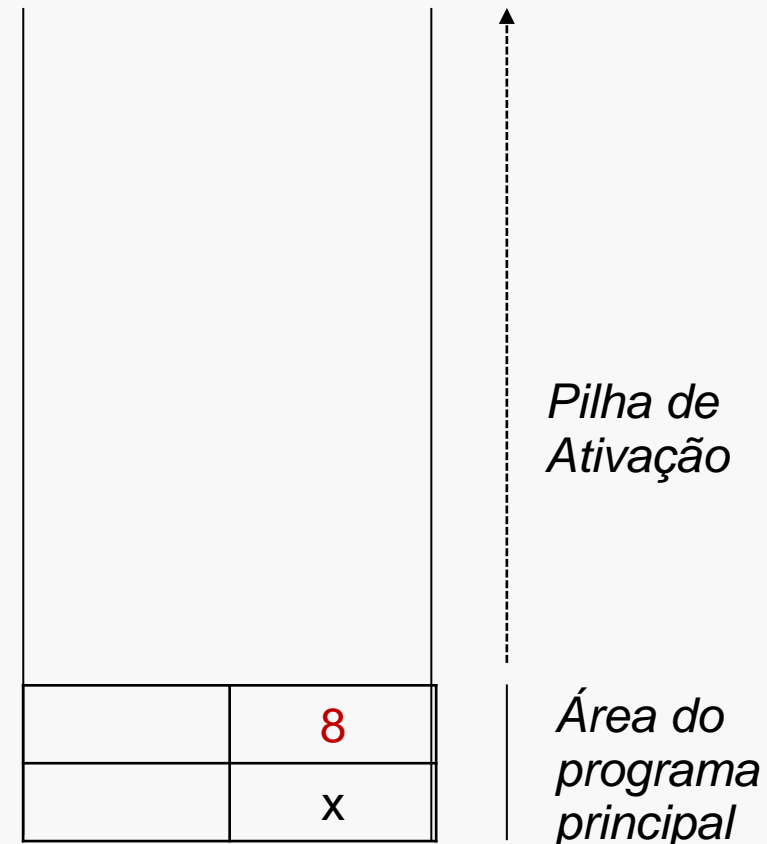
- Recursividade em Ação – Cálculo de 2^n
 - FIM DO PROGRAMA!

```
def pot2(n):  
    if n==0: return 1  
    else: return 2 * pot2(n-1)
```

```
x=pot2(3)  
print("2^3 = ",x) ←
```

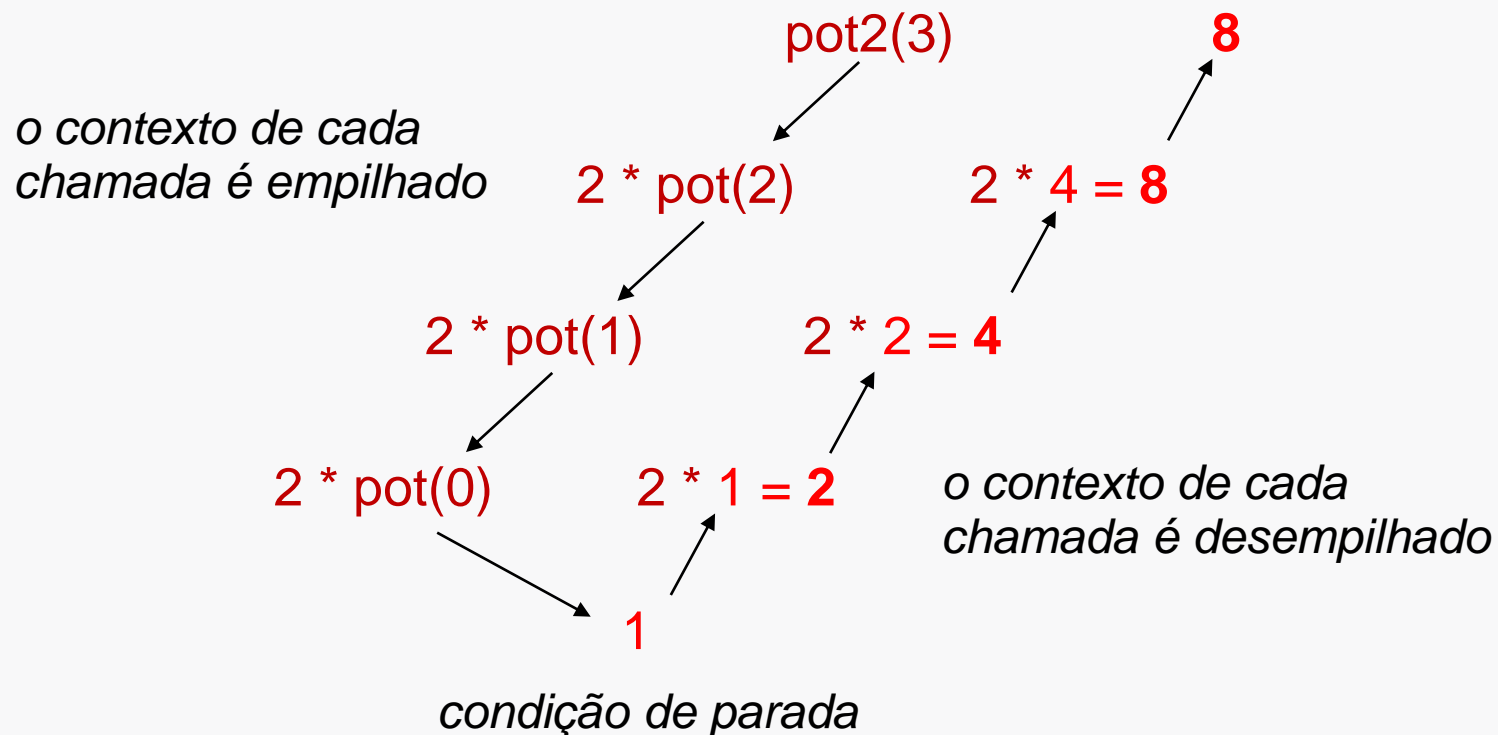
saída:

$2^3 = 8$



Recursividade em Python (19/19)

- **Árvore das Chamadas de Função – Cálculo de 2^n**
 - Podemos representar graficamente cada chamada da função “**pot2**” em uma estrutura chamada árvore de recursão.
 - A representação ajuda a entender o funcionamento da função.



Outros Exemplos (1/10)

- **Exemplo 2:** Cálculo de $s = (1/1) + (3/2) + (5/3) + (7/4) + \dots + (99/50)$
 - Assim como no exemplo da potência de 2, a recursividade substitui o uso de um comando de repetição (for ou while).

```
2 def calc_serie(n, d):
3     #print("{} / {}".format(n,d))
4     if n==1: return 1
5     else:
6         return n/d + calc_serie(n-2,d-1)
7
8 s = calc_serie(99,50)
9 print(s)
10
```

Shell ×

```
>>> %Run p03_serie.py
```

```
95.5007946616706
```

Outros Exemplos (2/10)

- Exemplo 3: Encontrar o maior valor em um vetor ou lista não-ordenada.

```

1 #p04_encontrar maior valor em vetor
2 def maior (n, v):
3     if (n == 1):
4         return v[0]
5     else:
6         #print("maior({},v)".format(n-1))
7         x = maior(n-1,v)
8         if (x > v[n-1]):
9             return x
10        else:
11            return v[n-1]
12
13 v = [5, 4, 10, -7]
14 x = maior(len(v),v)
15 print(x)
16

```

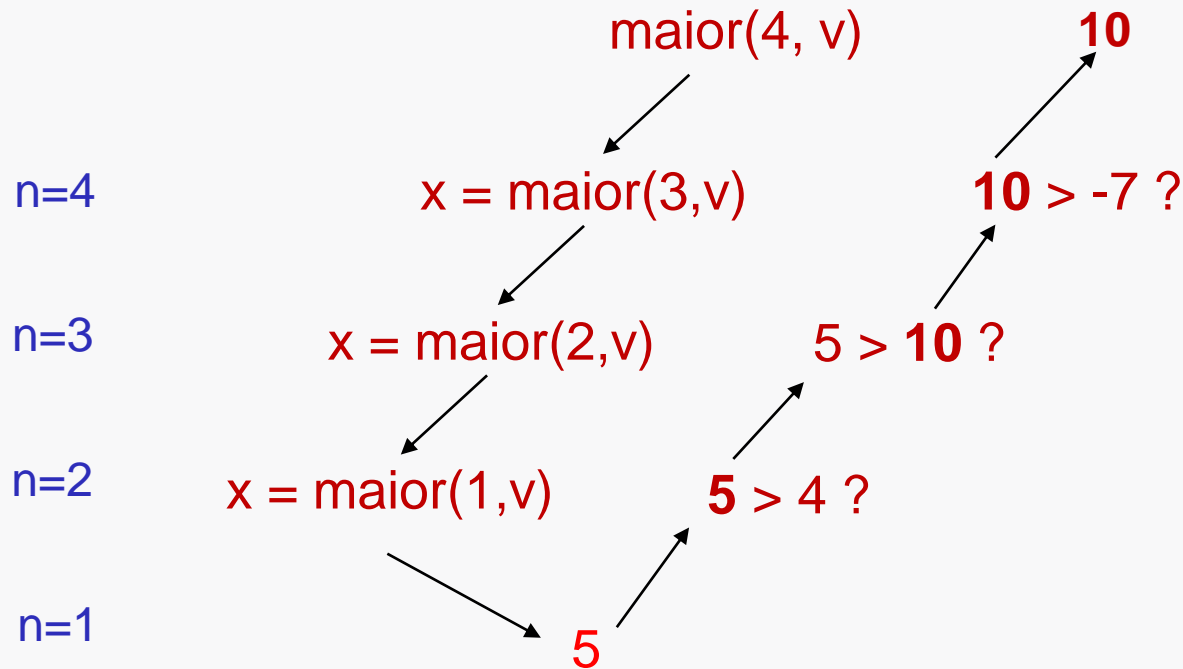
hell x

```
>> %Run p04_maior_valor_vetor.py
```

10

Outros Exemplos (3/10)

- Exemplo 3: Encontrar o maior valor em um vetor ou lista não-ordenada.
 - Árvore das chamadas, p/ $v = [5, 4, 10, -7]$



Outros Exemplos (4/10)

- **Exemplo 4: Sequência de Fibonacci**

- A Sequência de Fibonacci tem como primeiros termos os números 0 e 1 e, a seguir, cada termo subsequente é obtido pela soma dos dois termos predecessores:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...


- Os números de Fibonacci podem ser gerados através da seguinte definição recursiva:

$$\text{Fibonacci}(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2), & \text{se } n > 1 \end{cases}$$

Outros Exemplos (5/10)

- Exemplo 4: Sequência de Fibonacci

- A implementação de uma função recursiva em Python (ou qualquer linguagem) é uma tradução direta!

Fibonacci(n) 

- 0, se $n = 0$
- 1, se $n = 1$
- Fibonacci($n-1$) + Fibonacci($n-2$), se $n > 1$

```
1 def fib(n):
2     #casos base
3     if n == 0: return 0
4     elif n == 1: return 1
5     #caso recursivo
6     else: return fib(n-1) + fib(n-2)
7
8 x = fib(10)
9 print(x)
```

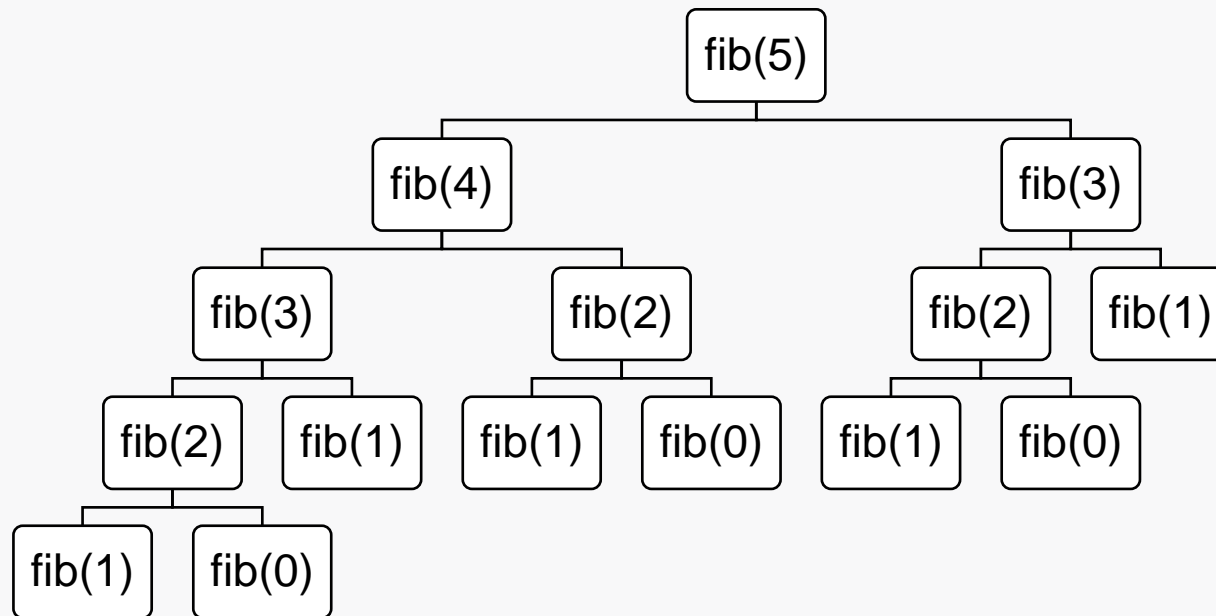
Shell ×

```
>>> %Run p05_fibonacci.py
```

```
55
```


Outros Exemplos (6/10)

- Seq. de Fibonacci – **Problemas** da Solução Recursiva Apresentada
 - Ela é **terrivelmente ineficiente**! Para $n > 1$, cada chamada causa 2 novas chamadas da função.
 - Veja no diagrama que para **fib(5)**, são feitas 15 chamadas (sendo 14 delas recursivas).
 - Veja ainda que **fib(1)** é chamada 5 vezes; **fib(0)** e **fib(2)** 3 vezes; **fib(3)**, 2 vezes.



- De fato, o número de chamadas **cresce exponencialmente** (complexidade exponencial). **Ex.:** para **fib(50)** são feitas 40.730.022.147 chamadas recursivas!

Outros Exemplos (7/10)

- Exemplo 5: Busca Binária

- Algoritmo para **buscar um valor** em uma lista ou vetor **ordenado**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

- Só pode ser usado se o vetor estiver ordenado. Caso contrário, temos que fazer uma **busca sequencial**
 - Significa percorrer os n elementos – complexidade $O(n)$.
- É um dos mais importantes algoritmos da ciência da computação e o motivo pelo qual frequentemente dados são guardados de forma ordenada.
 - Sua complexidade é **$O(\log n)$** .
 - Ou seja, ao contrário do Fibonacci recursivo “ingênuo” que acabamos de mostrar, a busca binária é **eficiente**!

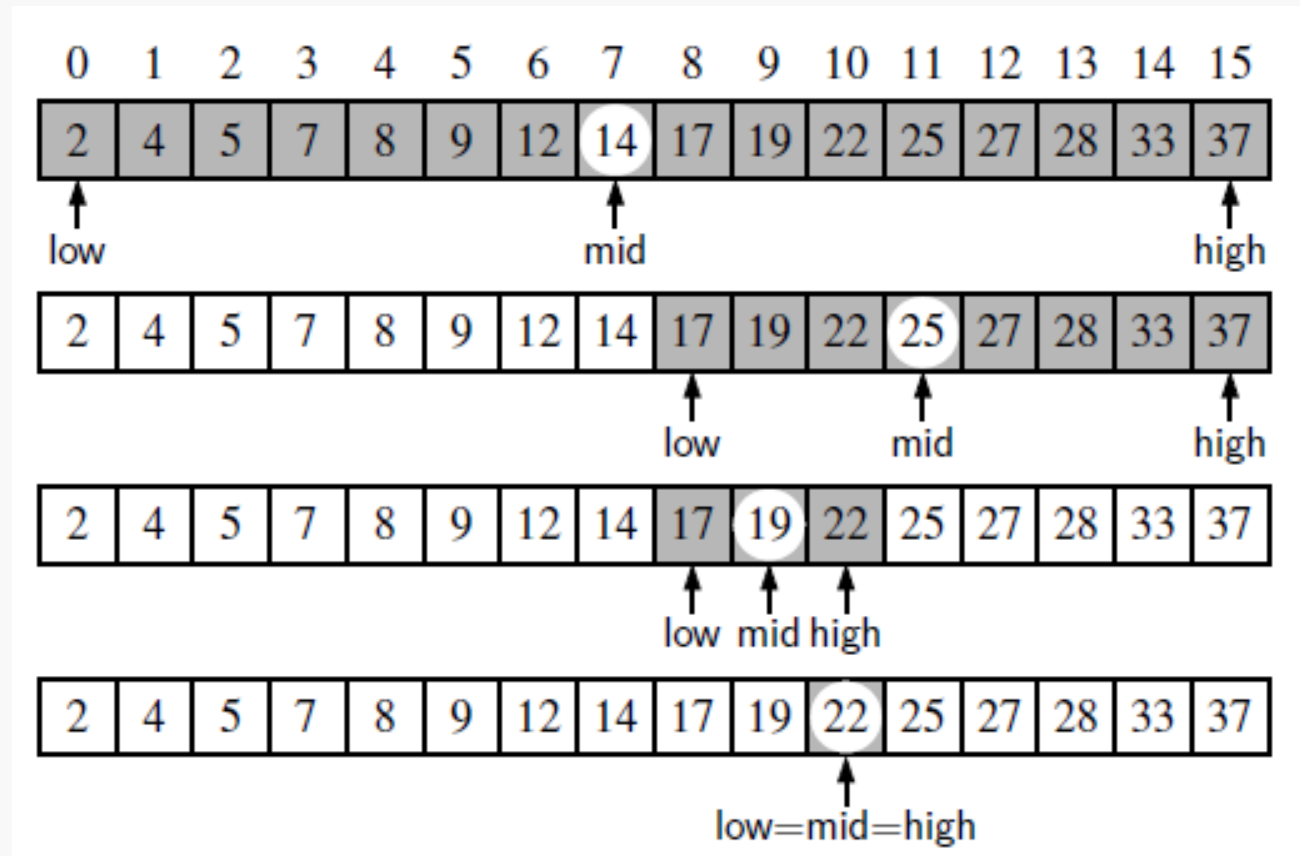
Outros Exemplos (8/10)

• Exemplo 5: Busca Binária Recursiva - Algoritmo

- Sejam: v um vetor ordenado com n elementos; e x o valor a ser buscado.
 - Inicialmente fazemos $low = 0$ e $high = n-1$
 - Então comparamos x com o elemento localizado na mediana dos valores localizados na sequência entre low e $high$.
 - $med = (low + high) // 2$
 - Consideramos então 3 casos:
 - Se $x = v[med]$: o elemento foi encontrado e a busca termina.
 - Se $x < v[med]$, então fazemos uma chamada recursiva na primeira metade da sequência (entre low e $med-1$).
 - Se $x > v[med]$, então fazemos uma chamada recursiva na segunda metade da sequência (entre $med+1$ e $high$).
 - Uma busca mal sucedida ocorrerá se $low > high$ (algoritmo deve parar, pois elemento não foi encontrado).

Outros Exemplos (9/10)

- Exemplo 5: Busca Binária – Exemplo: busca pelo valor 22



Outros Exemplos (10/10)

- Exemplo 5: Busca Binária

```

1 #p06: Busca Binária
2 def busca_binaria(v, x, low, high):
3     if low > high: return None #valor não encontrado
4     else:
5         med = (low + high) // 2
6         if x == v[med]: return med #achou o elemento
7         elif x < v[med]: return busca_binaria(v, x, 0, med-1)
8         else: return busca_binaria(v, x, med+1, high)
9
10 v = [2,4,5,7,8,9,12,14,17,19,22,25,27,28,33,37]
11 x=22
12 i = busca_binaria(v, x, 0, len(v)-1)
13 print("posição de {} = {}".format(x,i))
14

```

hell x

```

>> %Run p06_busca_binaria.py
posição de 22 = 10

```

Considerações Adicionais (1/5)

- **Recursividade Mútua**
 - Ocorre quando a recursividade se dá através de duas funções que se chamam reciprocamente.
 - **Ex.:** a função “ding”, chama a função “dong” que, por sua vez chama “ding”.
 - Esse tipo de recursão é também chamado de **recursão indireta**.

```
1 #recursividade mútua
2 def ding(n):
3     print("ding")
4     if n>1: dong(n-1)
5
6 def dong(n):
7     print("dong")
8     if n>1: ding(n-1)
9
10 #programa principal
11 ding(4)
```

Shell ×

```
>>> %Run p06_ding_dong.py
ding
dong
ding
dong
```

Considerações Adicionais (2/5)

• Recursividade como Repetição

- A recursividade pode ser também considerada uma forma de **repetição** de um determinado trecho de código
- Por exemplo, na função “**pot2**”, apesar de nenhum comando explícito de repetição ter sido utilizado, na prática o código executa um produto de n termos.

```

1  #função recursiva para calcular 2^n
2  def pot2(n):
3      if n==0: return 1
4      else: return 2 * pot2(n-1)
5
6
7  x=pot2(3)
8  print("2^3 = ",x)

```

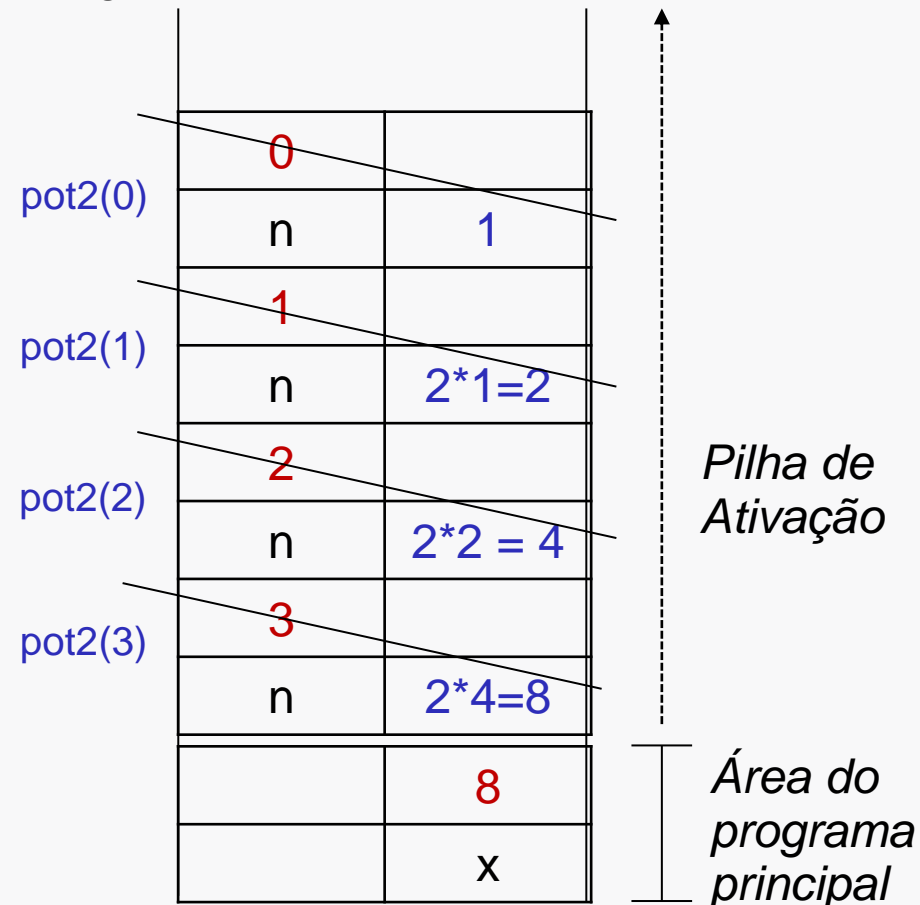
Shell ×

```

>>> %Run p02_potencia_de_2.py

2^3 = 8

```



Considerações Adicionais (3/5)

• Algoritmo Iterativo

- Para todo algoritmo recursivo, existe um outro equivalente **iterativo** (não recursivo).
- No entanto, problemas de natureza recursiva, costumam ser resolvidos de forma mais compacta e elegante através de funções recursivas.

```

1 #função recursiva para calcular 2^n
2 def pot2(n):
3     if n==0: return 1
4     else: return 2 * pot2(n-1)
5
6
7 x=pot2(3)
8 print("2^3 = ",x)

```

Shell ×

```

>>> %Run p02_potencia_de_2.py
2^3 = 8

```

```

1 #função iterativa para calcular 2^n
2 def pot2(n):
3     resultado = 1
4     for i in range(n):
5         resultado *= 2
6
7     return resultado
8
9
10 x=pot2(3)
11 print("2^3 = ",x)

```

Shell ×

```

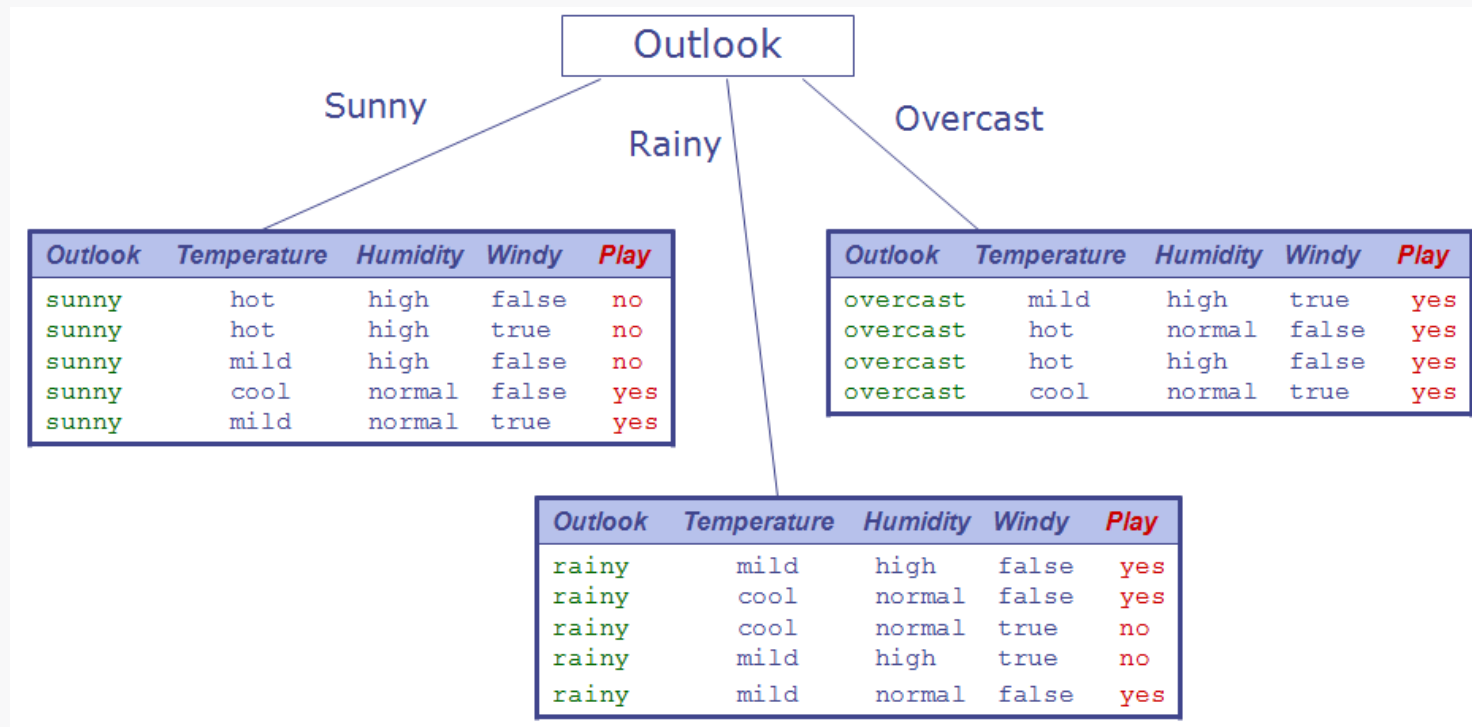
>>> %Run p07_pot2_iterativo.py
2^3 = 8

```


Considerações Adicionais (4/5)

- **Recursividade e Ciência de Dados**

- Há um bom número de algoritmos de ciência de dados que costumam ser implementados com o uso de recursividade.
- Um exemplo são os algoritmos de **árvores de decisão** para classificação e regressão



Considerações Adicionais (5/5)

- **Vantagens da Recursão**

- Para um problema recursivo por natureza, proporciona uma solução mais natural e compacta.
- Mais importante: para **certas categorias de problemas**, como **busca em árvore** e algoritmos do tipo **dividir para conquistar**, trabalhar com soluções não recursivas é tão difícil que se torna quase inviável.

- **Desvantagens da Recursão**

- Algoritmos recursivos costumam **consumir mais recursos**. O uso intensivo da pilha de ativação, pode fazê-la “estourar” (acabar memória disponível para empilhar).
- A recursão é **ineficiente** para muitos problemas (ex.: Fibonacci “ingênuo”, onde vários cálculos desnecessários são feitos, resultando em solução mais lenta).
- Algoritmos recursivos são mais difíceis de serem depurados, especialmente quando há vários níveis de recursão.

Referências

- Goodrich, M. T., Tamassia, R. e Goldwasser, M. H. (2013). “Data Structures and Algorithms in Python”. Wiley (*Cap. 4*).
- Corbucci, D. e Fernandes L. A. F. (2019) “Aula 04 - Funções, Parâmetros, Recursão”. Disponível em:
<http://www.ic.uff.br/~fabio/python.htm>
- Python 3 Tutorial – Recursion and Recursive Functions.
https://www.python-course.eu/python3_recursive_functions.php