





Algoritmos e Programação de Computadores

Funções Recursivas

Profa. Sandra Avila

Instituto de Computação (IC/Unicamp)

MC102, 12 Junho, 2019

Função Recursiva

- Uma função é chamada de recursiva **se o corpo da função chama a própria função**, direta ou indiretamente.
- Ou seja, o processo de execução do corpo de uma função recursiva pode, por sua vez, exigir a aplicação dessa função novamente.
- As funções recursivas não usam nenhuma sintaxe especial em Python, mas requerem algum esforço para entender e criar.

Soma de Números

- Vamos começar a nossa investigação com um problema simples que você já sabe como resolver sem o uso de recursão.
- Suponha que você deseja calcular a soma de uma lista de números, tais como: `[1, 3, 5, 7, 9]`.

Soma de Números

- Uma função iterativa que calcula a soma é mostrada a seguir.
- A função usa uma variável acumuladora (`soma`) para calcular o total de todos os números da lista iniciando com 0 e somando cada número da lista.

```
def somaLista (numeros) :  
    soma = 0  
    for i in numeros:  
        soma = soma + i  
    return soma
```

Soma de Números

- Uma função iterativa que calcula a soma é mostrada a seguir.
- A função usa uma variável acumuladora (`soma`) para calcular o total de todos os números da lista iniciando com 0 e somando cada número da lista.

```
def somaLista (numeros) :  
    soma = 0  
    for i in numeros:  
        soma = soma + i  
    return soma
```

```
lista = [1,3,5,7,9]  
print(somaLista(lista))
```

Soma de Números

- Imagine que você **não** tem laços `while` ou `for`.
- Como você calcularia a soma de uma lista de números?

Soma de Números

- Imagine que você **não** tem laços `while` ou `for`.
- Como você calcularia a soma de uma lista de números?
- Se você fosse um matemático poderia começar recordando que a adição é uma função definida para dois parâmetros, um par de números.

Soma de Números

- Para redefinir o problema da adição de uma lista para a adição de pares de números, podemos reescrever a lista como uma expressão totalmente entre parênteses.
- Tal expressão poderia ser algo como: $((((1+3) +5) +7) +9)$
- Poderíamos também colocar os parênteses na ordem reversa, $(1+ (3+ (5+ (7+9))))$

Soma de Números

- Observe que o par de parênteses mais interno, $(7+9)$, é um problema que podemos resolver sem um laço ou qualquer construção especial.
- Na verdade, pode-se utilizar a seguinte sequência de simplificações para calcular uma soma final.

$$\text{total} = (1 + (3 + (5 + (7 + 9))))$$

$$\text{total} = (1 + (3 + (5 + 16)))$$

$$\text{total} = (1 + (3 + 21))$$

$$\text{total} = (1 + 24)$$

$$\text{total} = 25$$

Soma de Números

- Como podemos usar essa ideia e transformá-la em um programa Python?

Soma de Números

- Como podemos usar essa ideia e transformá-la em um programa Python?
- Em primeiro lugar, vamos reformular o problema soma em termos de listas de Python.

Soma de Números

- Como podemos usar essa ideia e transformá-la em um programa Python?
- Em primeiro lugar, vamos reformular o problema soma em termos de listas de Python.
- Poderíamos dizer que a soma da lista `numeros` é a soma do primeiro elemento da lista (`numeros[0]`), com a soma dos números no resto da lista (`numeros[1:]`).

Soma de Números

- De forma funcional podemos escrever:

```
somaLista(numeros) = primeiro(numeros) + somaLista(resto(numeros))
```

- Nesta equação `primeiro(numeros)` retorna o primeiro elemento da lista e `resto(numeros)` retorna a lista com tudo menos o primeiro elemento.

Soma de Números

```
def somaLista(numeros):  
    if len(numeros) == 1:  
        return numeros[0]  
    else:  
        return numeros[0] + somaLista(numeros[1:])
```

Soma de Números

```
def somaLista(numeros):  
    if len(numeros) == 1:  
        return numeros[0]  
    else:  
        return numeros[0] + somaLista(numeros[1:])
```

 **Recursão**

Soma de Números

```
def somaLista (numeros) :  
    if len (numeros) == 1 :  
        return numeros [0]  
    else :  
        return numeros [0] + somaLista (numeros [1:])
```

 **Recursão**

```
def somaLista (numeros) :  
    soma = 0  
    for i in numeros :  
        soma = soma + i  
    return soma
```

Fatorial

- Na matemática, o fatorial de um número natural n , representado por $n!$, é o produto de todos os inteiros positivos menores ou iguais a n .
- Exemplo: $4! = 4 * 3 * 2 * 1 = 24$

Fatorial

- Uma implementação natural usando uma declaração `while` acumula o total multiplicando juntos cada inteiro positivo até `n`.

```
def fatorial (n):  
    total, k = 1, 1  
    while k <= n:  
        total, k = total * k, k + 1  
    return total
```

Fatorial

- Uma implementação natural usando uma declaração `while` acumula o total multiplicando juntos cada inteiro positivo até `n`.

```
def fatorial (n):  
    total = 1  
    k = 1  
    while k <= n:  
        total = total * k  
        k = k + 1  
    return total
```

Fatorial

- Uma **implementação recursiva** de fatorial pode expressar o fatorial de n (ou $n!$) em termos do fatorial de $n-1$, isto é, $(n-1)!$, e o caso base da recursão é a forma mais simples do problema: $1! = 1$.

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

Fatorial

- Essas duas funções fatoriais diferem conceitualmente.
- A **função iterativa** constrói o resultado a partir do caso base 1, para o total final, multiplicando-se sucessivamente cada termo.
- A **função recursiva**, por outro lado, constrói o resultado diretamente do termo final, n , e o resultado do problema mais simples, fatorial $(n-1)$.

Recursão com várias chamadas

- Não há necessidade da função recursiva ter apenas uma chamada para si própria.
- A função pode fazer várias chamadas para si própria.
- A função pode ainda fazer chamadas recursivas indiretas. Neste caso a função 1, por exemplo, chama uma outra função 2 que por sua vez chama a função 1.

Números de Fibonacci

- A série de fibonacci é a seguinte:

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

Números de Fibonacci

- A série de fibonacci é a seguinte:

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

- Queremos determinar qual é o n-ésimo número da série que denotaremos por `fibonacci(n)`.
- Como descrever o n-ésimo número de fibonacci de forma recursiva?

Números de Fibonacci

- No caso base temos:
 - Se $n = 0$ então $\text{fibonacci}(0) = 0$.
 - Se $n = 1$ então $\text{fibonacci}(1) = 1$.

Números de Fibonacci

- No caso base temos:
 - Se $n = 0$ então $\text{fibonacci}(0) = 0$.
 - Se $n = 1$ então $\text{fibonacci}(1) = 1$.
- Sabendo casos anteriores podemos computar $\text{fibonacci}(n)$ como:
 - $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$.

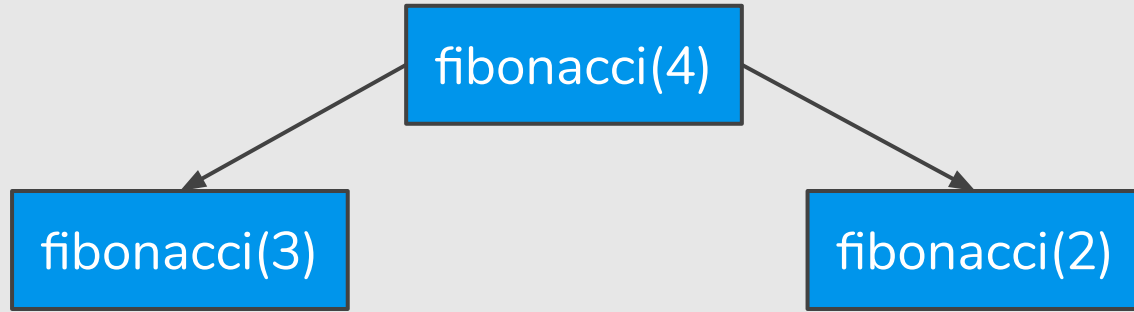
Números de Fibonacci

```
def fibonacci(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return (fibonacci(n-1) + fibonacci(n-2))
```

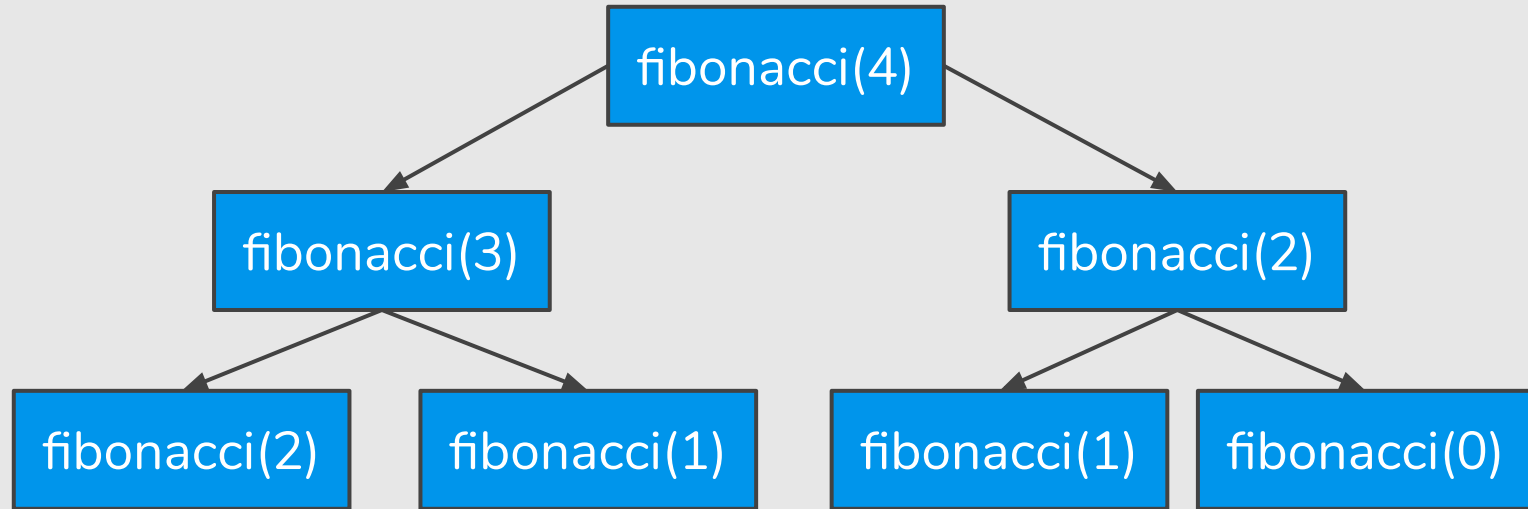
Números de Fibonacci

```
fibonacci(4)
```

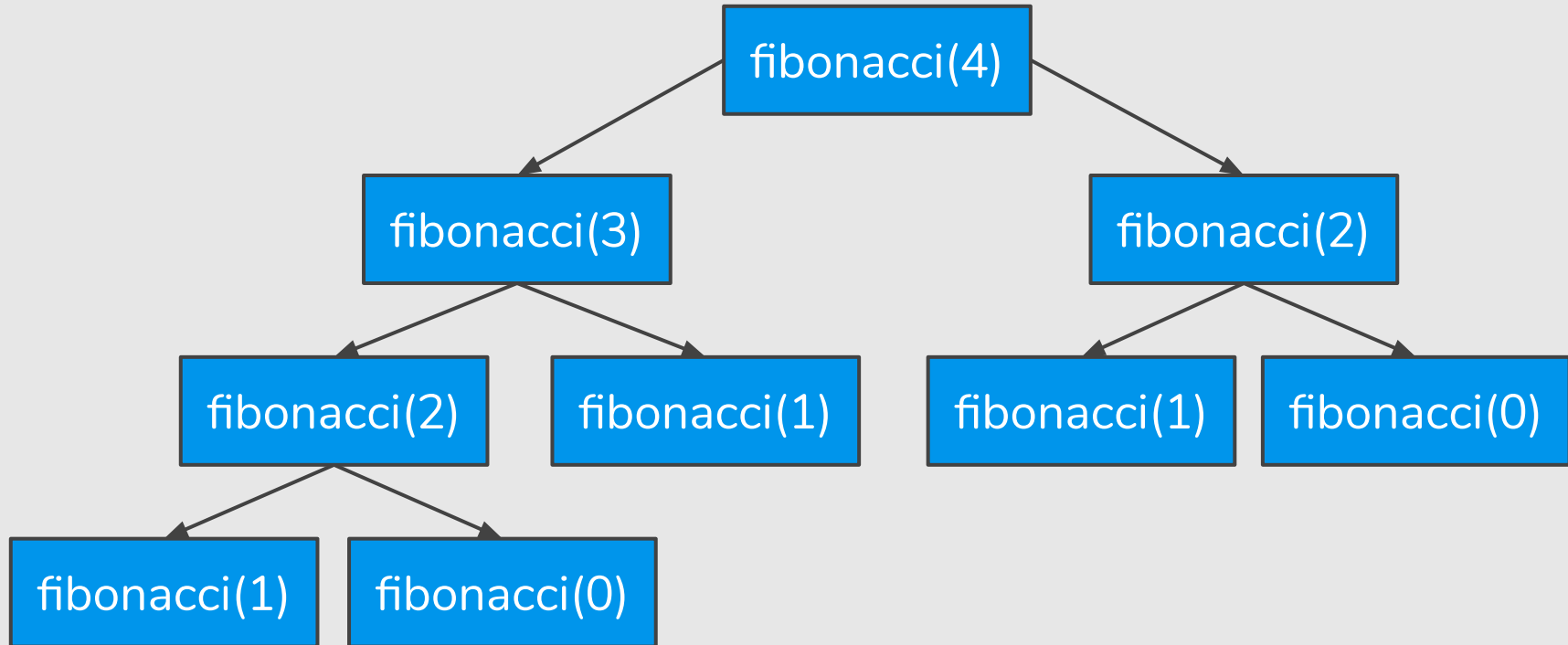
Números de Fibonacci



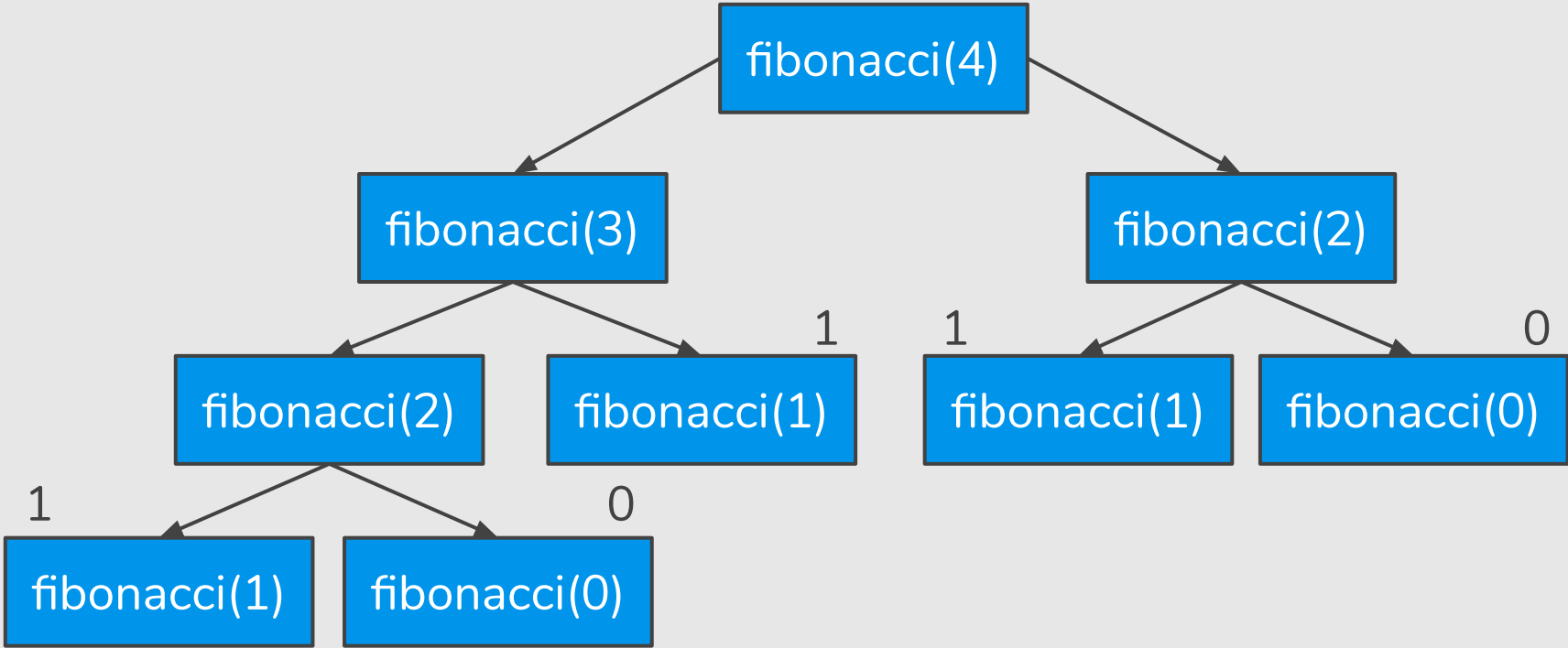
Números de Fibonacci



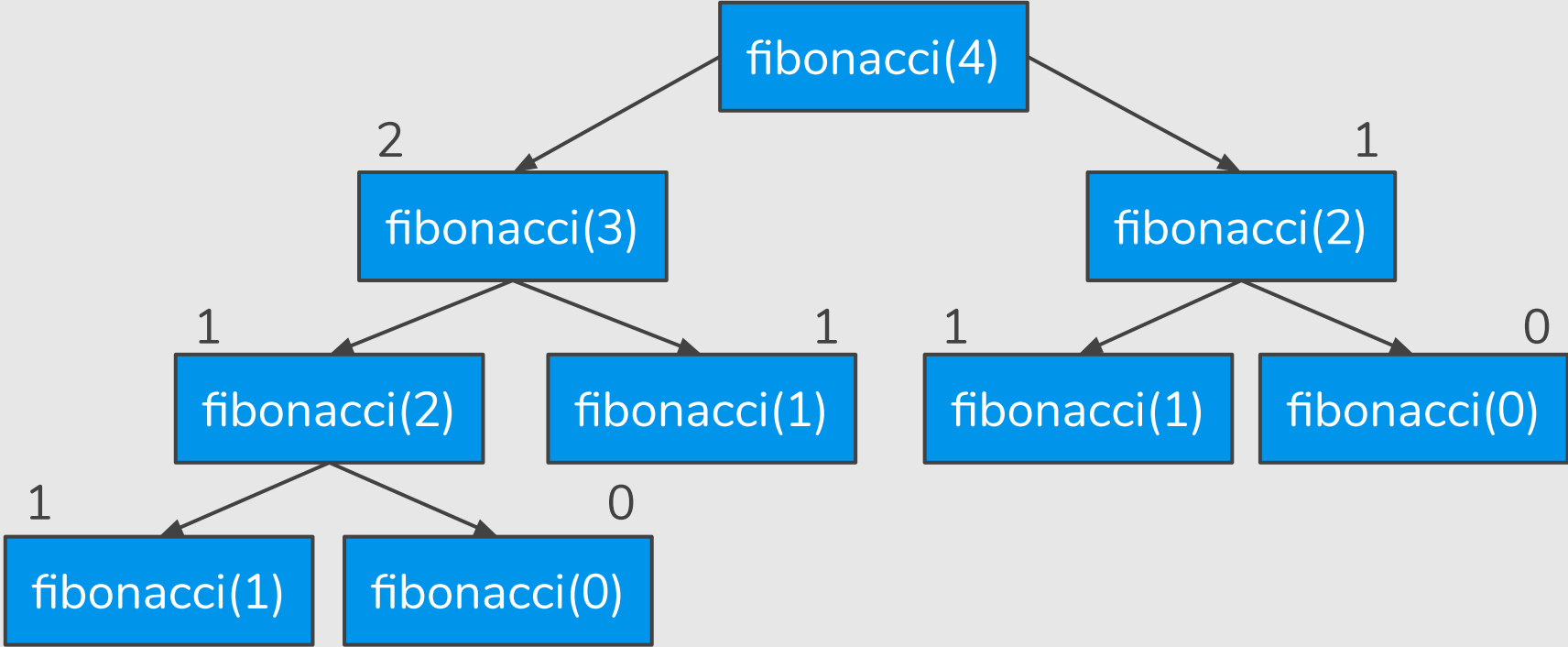
Números de Fibonacci



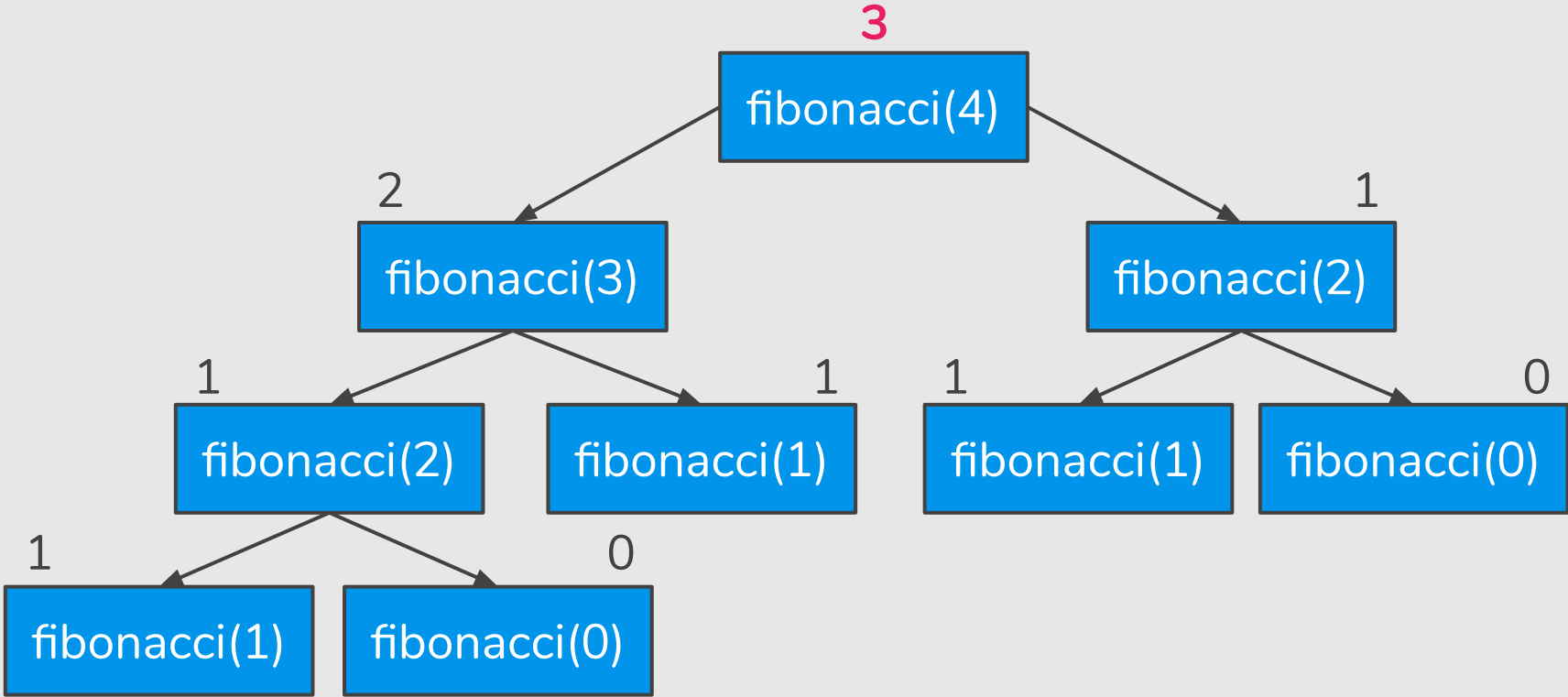
Números de Fibonacci



Números de Fibonacci



Números de Fibonacci



Resumindo ...

- Recursão é uma técnica para se criar algoritmos onde:
 - Devemos descrever soluções para casos básicos.
 - Assumindo a existência de soluções para casos menores, mostramos como obter a solução para o caso maior.
- Algoritmos recursivos geralmente são mais claros e concisos.
- Implementador deve avaliar clareza de código × eficiência do algoritmo.

Referências

- Os slides dessa aula foram baseados no material de MC102 do Prof. Eduardo Xavier (IC/Unicamp).
- <https://panda.ime.usp.br/pensepy/static/pensepy/12-Recursao/recursionsimple-ptbr.html>