Programação I / Introdução à Programação

Capítulo 10, "Recursion"

João Pedro Pedroso

2024/2025

Última aula

- Resolução numérica de equações
- Dicionários: extenso de números em português
- Parâmetros opcionais
  `def f(x=2, y=3)`
- Instrução `assert`
  `assert x >= 0`
- Variáveis globais
- Variável "mágica" `__name__`
  `if __name__== "__main__":`

Hoje:

- Ordenação
- Recursão
- Séries de Fibonacci

Ordenação: listas

```
>>> a = [1, -20, 3, 7, 14, 0, -120]
>>> a.sort()
>>> a
[-120, -20, 0, 1, 3, 7, 14]
```

- como ordenar por uma ordem diferente?
  - por exemplo, pelo valor absoluto de cada número

Ordenação: funções auxiliares

- Utilizar função para definir ordenação:
  - parâmetro opcional `key=...` de `sort`

```
>>> a = [1, -20, 3, 7, 14, 0, -120]
>>> a.sort(key = abs)
>>> a
[0, 1, 3, 7, 14, -20, -120]
```

- Podemos definir função de ordenação:

```
def d(x):
    return abs(x - 10)
```

```
>>> a = [1, -20, 3, 7, 14, 0, -120]
>>> a.sort(key = d)
>>> a
[7, 14, 3, 1, 0, -20, -120]
```

Funções anónimas

- Podemos utilizar uma função anónima / *lambda function*
  - pequenas funções, que podem ser definidas com apenas uma expressão
  - podem ser usadas como outras funções

```
>>> g = lambda x: x**2
>>> g(3)
9
```

- Utilização na ordenação:

```
>>> a = [1, -20, 3, 7, 14, 0, -120]
>>> a.sort(key = lambda x: abs(x-200))
>>> a
[14, 7, 3, 1, 0, -20, -120]
```

- Exemplo de função anónima com dois argumentos:

```
>>> f = lambda x,y: x+y
>>> f(1,2)
3
```

Obter dicionários ordenados por valor

```
>>> x = {1: "um", 3: "três", 4: "quatro", 2: "dois", 0: "zero"}
>>> for k in x:
        print(k, x[k])
1 um
3 três
4 quatro
2 dois
0 zero
>>> x.get(2)
'dois'
>>> for k in sorted(x, key=x.get):
        print(k,x[k])
2 dois
4 quatro
3 três
1 um
0 zero
```

Iteração e Recursão

Iteração e Recursão

Iteração repetição de uma sequência de operações

Recursão solução de um problema através de problemas semelhantes mas mais pequenos

Notas:

- Tanto a iteração como a recursão envolvem repetição e uma condição de paragem
  - a cada repetição, aproximamo-nos da condição de paragem
  - a iteração termina quando a condição de continuação do ciclo falha
  - a recursão termina quando se atinge um caso base
- Pode-se converter iteração em recursão, e vice-versa
  - chamadas recursivas consomem mais tempo e memória

Recursividade

Recursividade: definição de função, relação ou processo em termos de si próprio.

Exemplos:

antepassado pai ou mãe ou um dos seus antepassados

diretório estrutura que contém ficheiros e diretórios

árvore uma folha, ou um ramo que bifurca em duas ou mais árvores

Exemplo: Fatorial

Definição iterativa:

$$n! = n \times (n-1) \times \ldots \times 1$$

Definição recursiva:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \qquad (n > 0) \end{aligned}$$

- A primeira equação define o caso base (0!)
- A segunda equação define o caso recursivo: $n!$ usando $(n-1)!$
- Fatorial fica definido para *todos* os inteiros não-negativos

Em concreto:

$$
\begin{aligned}
4! &= 4 \times 3! \\
&= 4 \times (3 \times 2!) \\
&= 4 \times (3 \times (2 \times 1!)) \\
&= 4 \times (3 \times (2 \times (1 \times 0!))) \\
&= 4 \times (3 \times (2 \times (1 \times 1))) \\
&= 24
\end{aligned}
$$

Em Python:

```
def fatorial(n):
    if n == 0:
        r = 1  # caso base
    else:
        r = n * fatorial(n-1)  # caso recursivo
    return r
```

Ver exemplo de execução no Python Tutor

Sistematizando:

Uma função recursiva envolve a definição de:

- um ou mais casos base, para os quais a solução é conhecida
- caso recursivo
  - usamos a mesma função para resolver um problema mais simples
  - aproximamo-nos do caso base

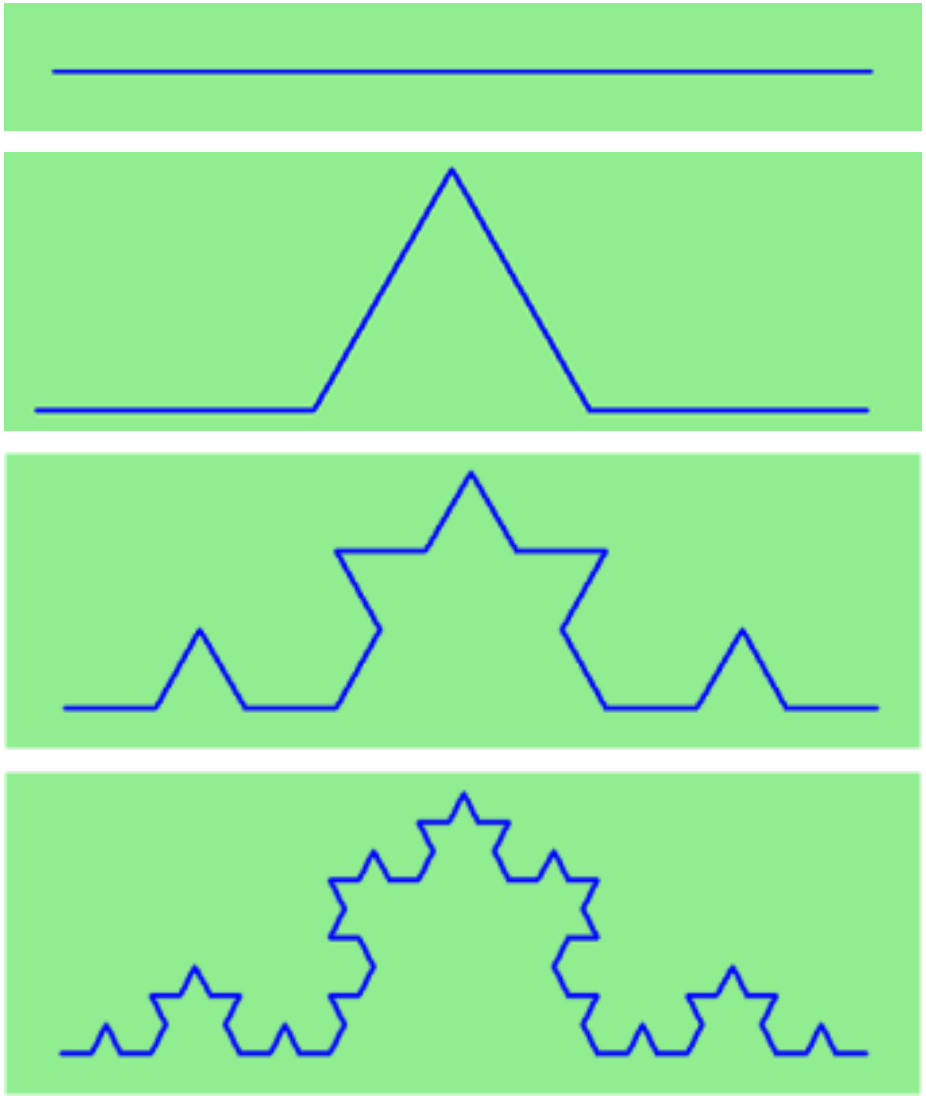
Exemplo: Fractal de Koch

$n = 0$

$n = 1$

$n = 2$

$n = 3$



- A curva de ordem 0 é uma linha
- A curva ordem $n + 1$ consiste em quatro curvas de ordem n

Em Turtle:

```
from turtle import *

def koch(n, size):
    if n == 0:                  # caso base: uma linha
        forward(size)
    else:                       # caso recursivo
        koch(n-1, size/3)       # 4 curvas com 1/3 do comprimento
        left(60)
        koch(n-1, size/3)
        right(120)
        koch(n-1, size/3)
        left(60)
        koch(n-1, size/3)
```

Simplificando:

- `right(`α`)` é equivalente a `left(`$-\alpha$`)`
- `left(0)` não altera a orientação

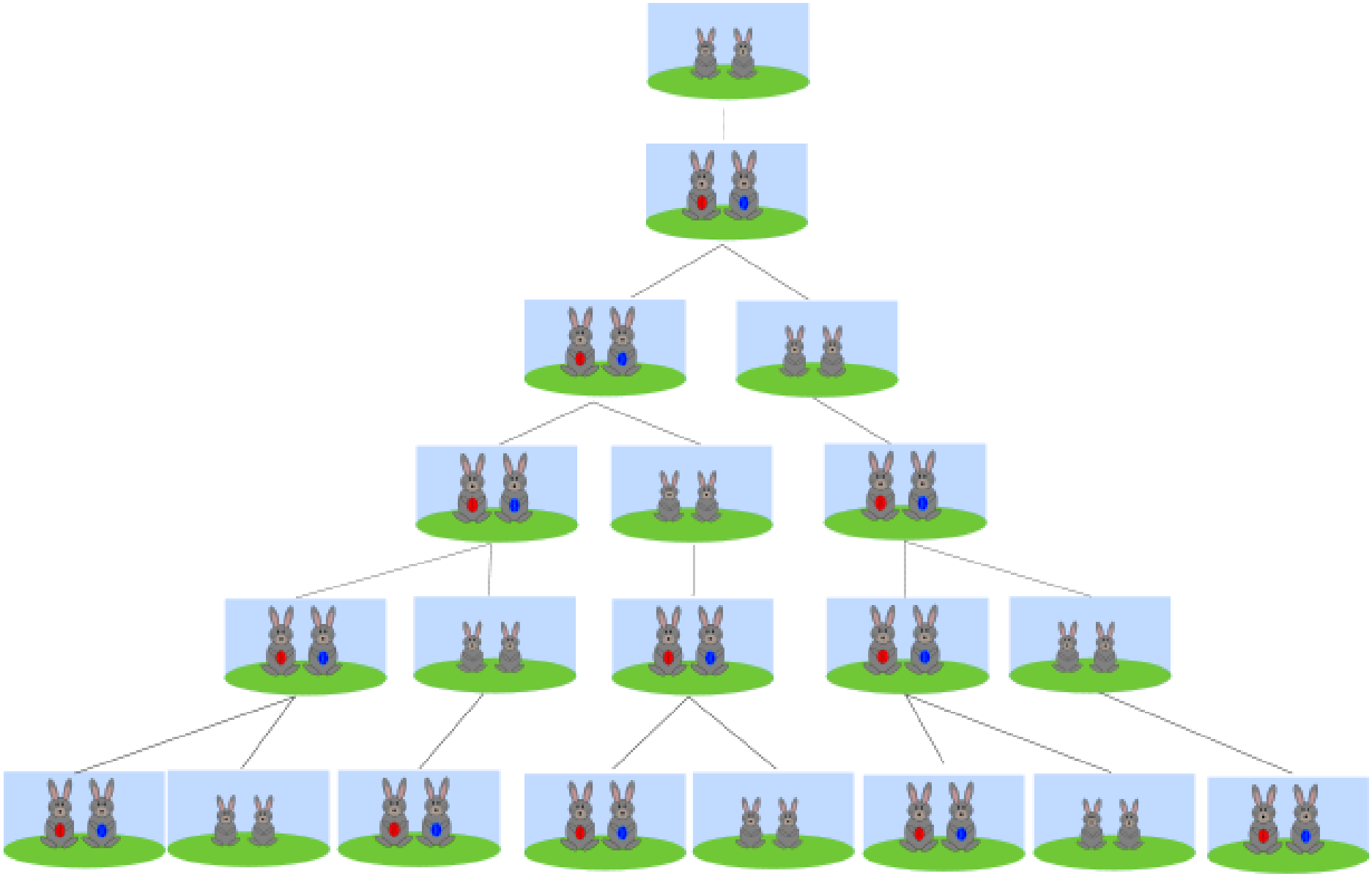
```
def koch(n, size):
    if n == 0:                  # caso base: uma linha
        forward(size)
    else:                       # caso recursivo
        for angle in [60, -120, 60, 0]:
            koch(n-1, size/3) # avançar 1/3 do comprimento
            left(angle)
```

Série de Fibonnacci

Números de Fibonnacci

- Conhecidos na Índia (700 DC)
- Introduzidos na Europa por *Leonardo de Pisa* (1202 DC)
- Modelo simplificado para o crescimento de uma população de coelhos:
  - um par de coelhos são colocados num campo;
  - os coelhos podem acasalar com um mês;
  - no fim do segundo mês, nasce um novo par de coelhos;
  - os coelhos não morrem, e cada par produz sempre um novo par ao fim de um mês.
- Quantos pares de coelhos existem ao fim de um ano?

- Ao fim de 1 mês ainda há só 1 par.
- Ao fim de 2 meses nasce um novo par
  - no total, há 2 pares
- Ao fim de 3 meses, nasce um segundo par da primeira fêmea
  - no total, há 3 pares
- Ao fim de 4 meses, nasce o terceiro par da primeira fêmea e o primeiro par da segunda fémea
  - no total, há 5 pares



Ao fim de n meses, o número de pares é a soma de:

- número de pares que existiam no mês anterior
- número de pares que nascem
  - é o número de pares no mês $n - 2$

Simbolicamente

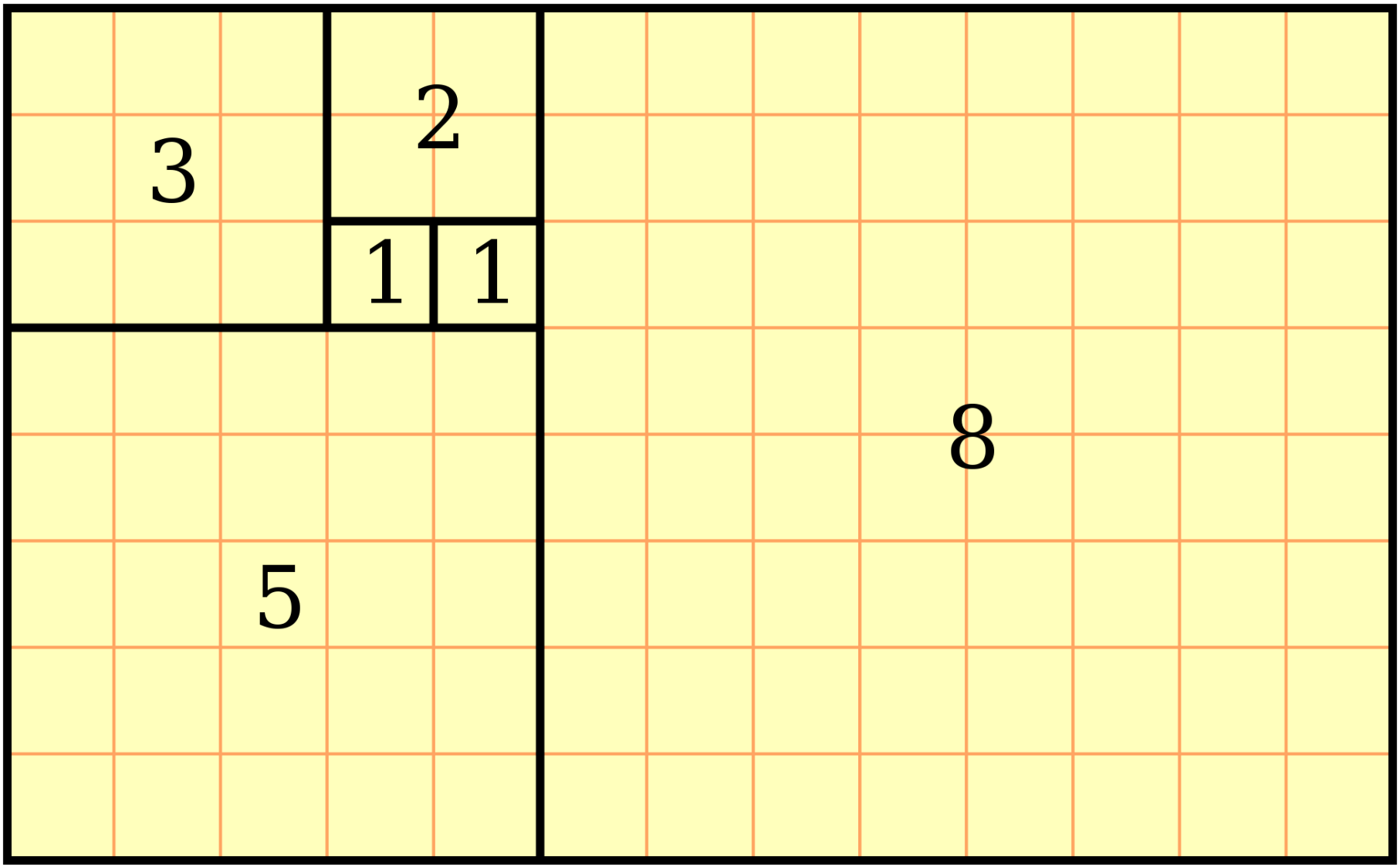
$$F_n = F_{n-1} + F_{n-2}$$

Casos base:

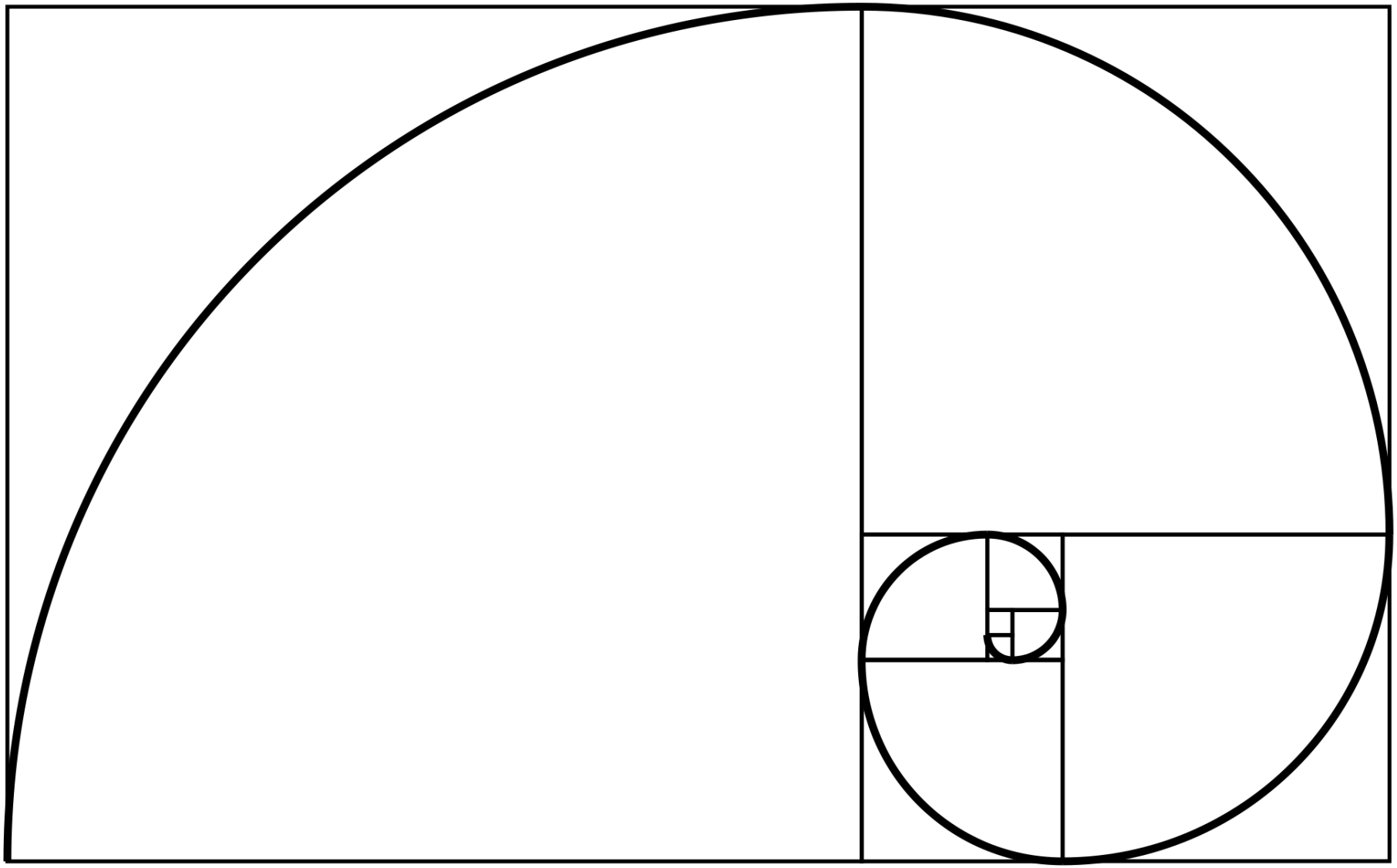
$$F_0 = 1$$

$$F_1 = 1$$

Série de Fibonacci: visualização com blocos



Série de Fibonacci: construção de espiral



Implementação recursiva

```
def fib(n):
    if n==0 or n==1:
        r=1
    else:
        r= fib(n-1) + fib(n-2)
    return r
```

Exemplo:

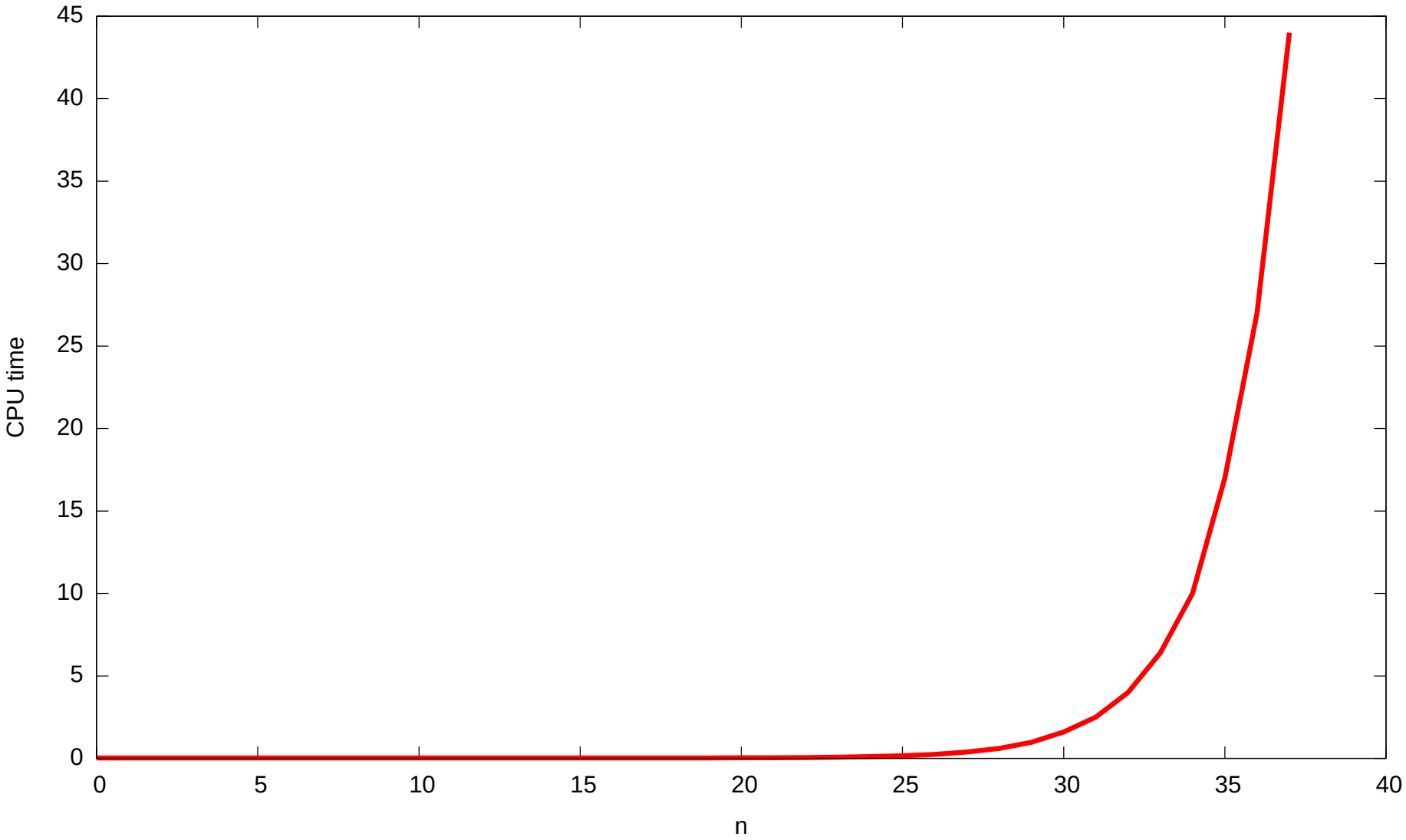
```
>>> fib(4)
5
```

Notas: esta implementação é pouco eficiente!

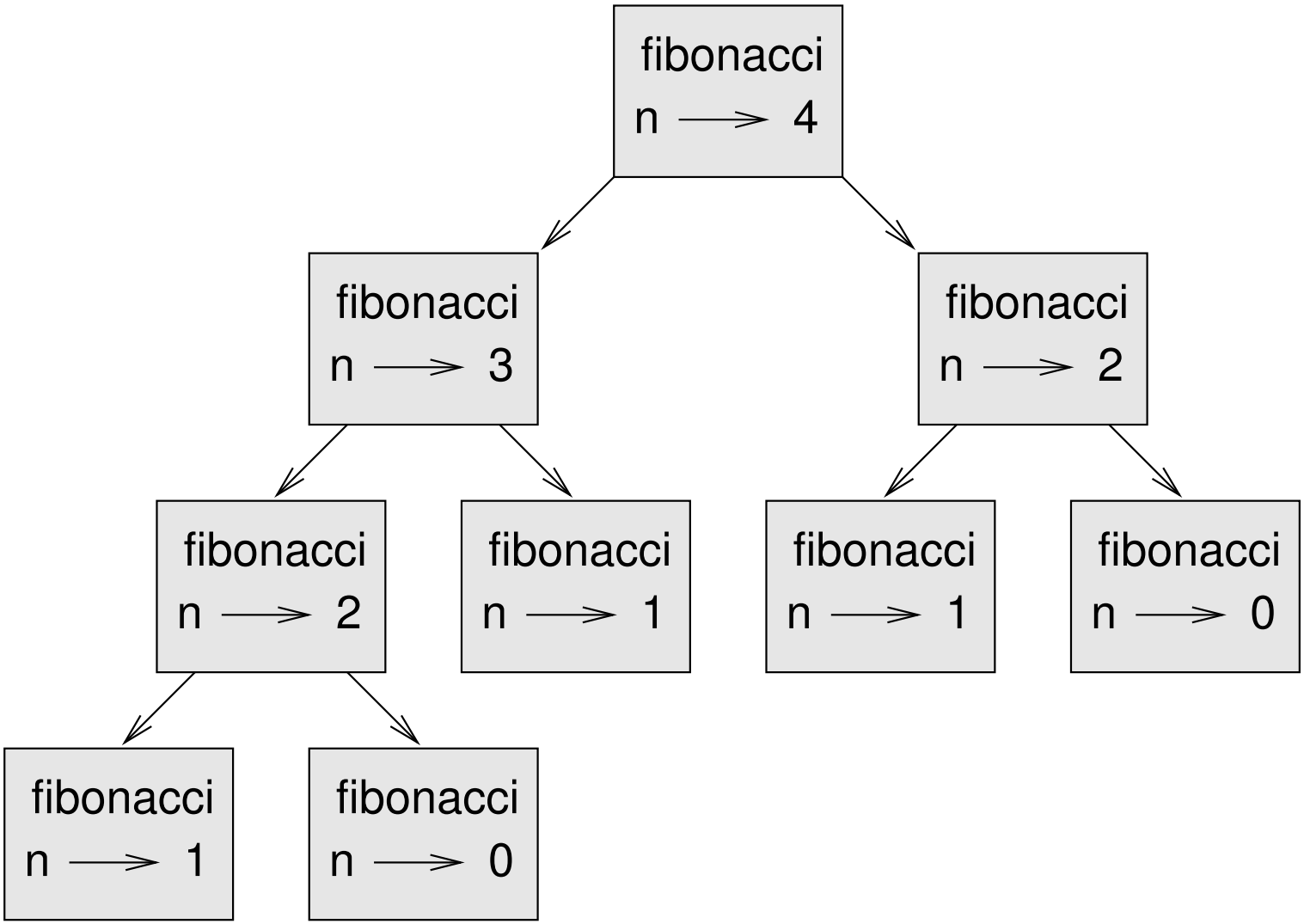
Eficiência

Construir uma tabela do tempo de computação em função de n.

```
import time
for n in range(1, 40):
    t0 = time.process_time()
    fn = fib(n)
    t1 = time.process_time()
    print("%d\t%f" % (n,t1-t0))
```


CPU time
n
0
5
10
15
20
25
30
35
40
45

Fibonacci: grafo de chamadas



A computação de `fib(2)` é repetida desnecessariamente...

Fibonacci com *memoização* (do inglês *memoization*)

Ideia: usar dicionário para guardar resultados já calculados

```
fib_memo = {0:1, 1:1} # dicionário global

def fastfib(n):
    if n in fib_memo:
        r = fib_memo[n]
    else:
        r = fastfib(n-1) + fastfib(n-2)
        fib_memo[n] =r
    return r
```

- Utilização (resultados praticamente instantâneos):

```
>>> fastfib(40)
165580141
>>> fastfib(500)
225591516161936330872512695036072072046011324913758190588638866418474627738686883405015987052796968498626L
```

Listas imbricadas

Listas imbricadas

Listas cujos elementos são:

- números inteiros ou vírgula flutuante
- outras listas imbricadas

Exemplos:

```
[1, 2, 3]
[1, [2, 3], 4]
[[1,2],[3,4],[[5],6]]
```

Como somar todos os elementos

- É necessário definir uma *função recursiva*

```
def recsum(xs):
    "Somar todos os valores numa lista imbricada."
    s = 0                       # acumulador da soma
    for x in xs:                # percorrer todos os elementos
        if type(x)==int or type(x)==float:
            s += x
        elif type(x)==list:
            s += recsum(x)  # somar sub-lista recursivamente
        else:
            raise TypeError("lista inválida")
    return s
```

Sumário das últimas aulas

Tipos nativos do Python:

- tipos simples: inteiros, floats, booleanos
- tipos compostos: strings, listas, tuplos, dicionários
  - listas, tuplos e dicionários podem ser *recursivos*

Próxima aula

- Utilização de ficheiros