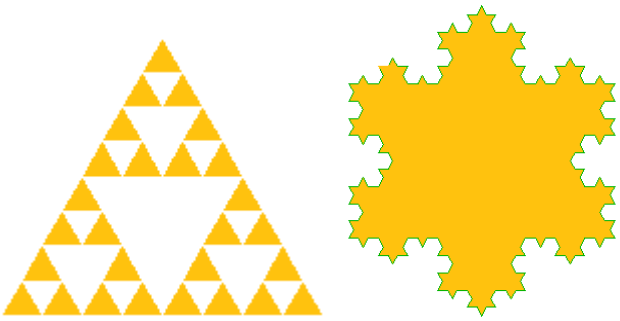
Capítulo

3 Recursividade



Objetivos

Os objetivos desse capítulo são os seguintes:

- Entender que problemas complexos que de outra forma seriam difíceis de resolver podem ter soluções recursivas simples.
- Aprender como formular programas recursivamente.
- Entender e aplicar as três leis da recursão.
- Entender recursão como uma forma de iteração.
- Implementar a formulação recursiva de um problema.
- Entender como recursão é implementada em um sistema computacional.

Recursão

A recursão ocorre quando um determinado procedimento ao ser executado faz referência ao próprio procedimento, isto é, para resolvermos o procedimento temos que chamar novamente este mesmo procedimento.

Um exemplo muito interessante está na construção do triângulo de Sierpinski[1] mostrado na figura 3.1. No processo de construção, primeiramente, pega-se um triângulo ligam-se os pontos médios de todos os seus lados obtendo-se assim, 3 novos triângulos. Em seguida para cada triângulo, repete-se o mesmo procedimento anterior, isto é, para cada triângulo obtido na etapa anterior, ligam-se os pontos médios de todos os seus lados obtendo-se assim, 3 novos triângulos.

Figura 3.1 Exemplo de recursividade no triângulo de Sierpinski.

O algoritmo 3.1 ilustra um procedimento para a construção do triângulo de Sierpinski, onde pode-se observar a passagem do parâmetro de entrada que é a figura geométrica triângulo. Na execução deste procedimento, ativa-se novamente o próprio procedimento denotando assim, uma chamada recursiva.

[1] Waclaw Sierpinski (1882 - 1969) foi um matemático polonês que primeiro descreveu este fractal.

```
Procedimento trianguloSierpinski(Ponto P1, Ponto P2, Ponto P3)
        Desenhar o Triângulo ( P1,P2,P3)
        // Obter os pontos médios
            P4 = (P1+P2)/2;
            P5 = (P2+P3)/2;
            P6 = (P1+P3)/2;
        trianguloSierpinski ( P1,P4,P6)
        trianguloSierpinski ( P4,P2,P5)
        trianguloSierpinski ( P6,P5,P3)
    fimSe
fimProcedimento
```

Algoritmo 3.1 Função recursiva para a construção do triângulo de Sierpinski sem os critérios de parada.

É importante frisar que este procedimento não terminaria nunca, pois, no seu corpo, não se faz nenhuma referência à uma condição de parada. Toda rotina recursiva tem que ter, um ou mais pontos de parada, que são os estágios que terminam a chamada recursiva.

Uma condição de parada que poderíamos colocar no Algoritmo 3.1, é o número de etapas de recursão, denominada de nível (**level**). Por exemplo, na figura 3.1, o número de etapas, ou o nível de profundidade de recursão é igual a 4. Denominando de level, o nível de profundidade de recursão e, considerando que este número seja passado como parâmetro, poderemos, neste caso, cria a condição de parada. Se o nível for igual a 1 a recursão é finalizada. O algoritmo 3.2, ilustra esta alteração. Na primeira vez que este procedimento for chamado, o nível é passado com parâmetro, e no corpo deste procedimento, a condição de parada é level = 0.

```
Procedimento trianguloSierpinski(int level,Ponto P1, Ponto P2, Ponto P3)
    Se level = 0   então  // Critério de parada
        Desenhar o Triângulo ( P1,P2,P3)
    senão
        // Obter os pontos médios
            P4 = (P1+P2)/2;
            P5 = (P2+P3)/2;
            P6 = (P1+P3)/2;
        trianguloSierpinski ( level-1,P1,P4,P6)
        trianguloSierpinski (level -1,P4,P2,P5)
        trianguloSierpinski (level -1,P6,P5,P3)
    fimse
Fim procedimento
```

Algoritmo 3.1 Função recursiva para a construção do triângulo de Sierpinski com critério de parada o nível.

O Fragmento de Código 3.2 implementa o algoritmo 3.1. O Fragmento de Código 3.3, implementa os métodos auxiliares drawTriangle que utiliza a biblioteca gráfica da classe turtle do Python e a função auxiliar **midlle_point** que retorna o ponto médio do segmento de reta cujos extremos são informados como parâmetros.

```
def sierpinski(P1,P2,P3,degree,myTurtle):
    colormap = ['blue','red','green','white','yellow',
                'violet','orange']
    if degree == 0:
        drawTriangle([P1,P2,P3],colormap[0],myTurtle)
    else:
        P4 = midlle_point(P1,P2)
        P5 = midlle_point(P2,P3)
        P6 = midlle_point(P1,P3)
        sierpinski(P1,P4,P6,degree-1, myTurtle)
        sierpinski(P4,P2,P5,degree-1, myTurtle)
        sierpinski(P6,P5,P3,degree-1, myTurtle)
```

Fragmento de Código 3.2 Implementação da função recursiva sierpinski para a construção do triângulo de Sierpinski e as funções auxiliares midlle_point para obter o ponto médio do segmento formado pelos pontos informados como parâmetros e drawTriangule utilizada para desenhar o triângulo.
Adaptado de
Disponível em http://cs4hs.cs.washington.edu/content/Resources/SessionMaterials/bin-o-slides/TeachingProgramming/Sierpinski.Python acessado em 20 de outubro de 2012.

```
import turtle
def drawTriangle(points,color,myTurtle):
    myTurtle.fillcolor(color)
    myTurtle.up()
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.down()
    myTurtle.begin_fill()
    myTurtle.goto(points[1][0],points[1][1])
    myTurtle.goto(points[2][0],points[2][1])
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.end_fill()

def midlle_point(p1,p2):
    return ( (p1[0]+p2[0]) / 2, (p1[1] + p2[1]) / 2)
```

Fragmento de Código 3.3 Implementação das funções auxiliares midlle_point para obter o ponto médio do segmento formado pelos pontos informados como parâmetros e drawTriangule utilizada para desenhar o triângulo.

Funções Recursivas

Uma função é dita ***recursiva*** quando na sua implementação utiliza-se uma chamada à própria função. A grande aplicação de recursividade está nas implementações das funções matemáticas. A estruturação de uma função na forma recursiva é a principal meta na elaboração dos algoritmos recursivos. Por exemplo, considere a definição da função fatorial a seguir.

O fatorial de um número natural n não nulo, representado por $n!$, é o produto do número por todos os seus antecessores naturais até o 1. Se $n = 0$, define-se o fatorial como 1, isto é $0! = 1$.

Pode-se observar, por exemplo, que pela definição anterior, escrevemos o fatorial de um número da seguinte forma

0! = 1
1! = 1
2! = 2.1
3! = 3.2.1
4! = 4.3.2.1
...

De forma genérica:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n.(n-1).(n-2)....1 & \text{se } n > 0 \end{cases}$$

Esta forma anterior é dita iterativa, pois existe uma iteração (repetição) de passos. O Fragmento de código, ilustra a implementação em Python da função *iterativeFactorial*, que calcula o fatorial de forma iterativa, isto é, sem utilizar o conceito de recursividade.

```
def iterativeFactorial(n):
    fat = 1
    for i in range(n,0,-1):
        fat = fat*i
    return fat
```

Fragmento de Código 3.1. Implementação em Python da função *iterativeFactorial* que calcula o fatorial de um número de forma iterativa.

Na definição recursiva do fatorial, vamos considerar que 0! = 1 é a solução ***trivial*** ou ***particular*** que será o caso base, e será o nosso critério de parada. Para $n > 0$, caso geral, teremos a chamada recursiva da função. Assim, podemos escrever:

1! = 1.0!
2! = 2.1!
3! = 3.2!
4! = 4.3!
...

De forma geral temos: $n! = n.(n-1)!$, para $n > 0$ (solução ***geral***) e 1 se $n = 0$ solução ***trivial*** ou ***particular***. Logo, a definição recursiva do fatorial será:

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

O fragmento de código 3.4, ilustra a implementação em Python da função *factorial* utilizando o conceito de recursividade. Pode-se observar que no caso de $n > 0$, faz-se a chamada novamente da função *factorial*, confirmando assim, como sendo uma função recursiva. O caso $n = 0$ é critério de parada.

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

Fragmento de Código 3.4. Implementação em Python da função que calcula o fatorial com recursão.

Se o valor do parâmetro n fosse igual a 3, a sequência de execução do Fragmento de Código 3.2 seria seguinte

```
factorial(3)
   factorial(2)
      factorial(1)
         factorial(0)
          return 1
       return 1*1 = 1
   return 2*1 = 2
return 3*2 = 6
```

O diagrama da Figura 3.1, ilustra a sequência de chamadas da função fatorial para o cálculo do fatorial de 3, representado por 3!, utilizando a forma recursiva. Pode-se observar que enquanto a rotina não chegar a ponto de parada, no caso $n = 0$, todas as chamadas das funções anteriores à função corrente, esperam o resultado da chamada anterior para enfim, retornar os seus respectivos valores. Este processo é denominado de *desempilhamento* da recursividade. Em toda função recursiva, a cada chamada de uma nova recursão, os parâmetros são guardados em uma estrutura de dados denominada de **pilha** que será estudada posteriormente. A cada término de uma recursão, os parâmetros são desempilhados da estrutura e utilizados no processo corrente.

Este processo é repetido até que se chegue na primeira função de ativação. A figura 3.2 ilustra esse procedimento.

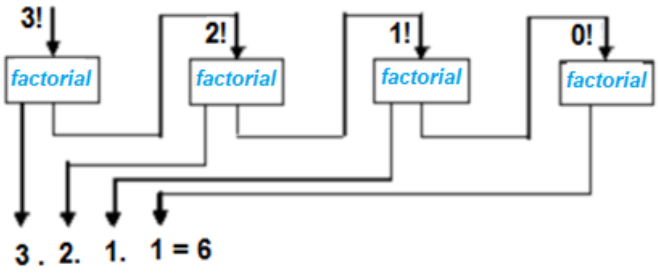


Figura 3.2 – Diagrama ilustrativo da execução da função recursiva do fatorial de 3.

Recursão Linear

A recursão linear é a forma mais simples de recursão, neste caso o tempo de máquina necessário para os processos recursivos cresce linearmente com o valor da entrada a processar. Esta recursão ocorre quando, para uma determinada entrada, somente uma chamada recursiva da própria função é executada. A recursividade é então ativada até que um determinado critério de parada seja alcançado.

```
def recfunc(parameters):
      <casos base>
      ...
      recfunc(parameters)
      ...
```

A função fatorial do Fragmento de Código 3.2 ilustra uma recursão linear, onde pode-se observar, no corpo da função, a ativação de uma chamada à função fatorial somente uma vez, logo para o tamnho da entrada igual a n , teremos n chamada recursivas da função.

Um outro exemplo de recursão linear é a função MDC, utilizada para o cálculo do máximo divisor comum entre dois números naturais. Uma forma recursiva de se representar o MDC entre dois números inteiros x e y pode ser escrita

$$MDC(x,y) = \begin{cases} x \ para & x = y \\ MDC(y,x) \ para & x < y \\ MDC(x-y,y) \ para & x > y \end{cases}$$

Por exemplo, para calcularmos o MDC entre 3 e 6, a sequência de passos será:

$$MDC(3,6) = MDC(6,3) = MDC(6-3,3) = MDC(3,3) = 3$$

Para afirmar que uma função possui uma recursão linear é preciso verificar que para qualquer entrada de ativação da recursão, esta função só poderá ser ativada uma única vez.

O Fragmento de código 3.5 ilustra a função de recursão linear MDC, com os números inteiros informados como parâmetros. Pode-se observar que, no corpo da função, existem para cada entrada, somente uma nova chamada da função recursiva, isto é, se x < y a função MDC é chamada novamente somente uma vez e, no caso x > y, também se faz uma única chamada da função MDC.

```
def MDC(x,y):
    if (x == y):
        return x
    elif x < y:
        return MDC(y,x)
    else:
        return MDC(x-y,y)
```

Fragmento de Código 3.5. Implementação em Python da função MDC com recursão.

Recursão Binária

Esta recursão ocorre quando o tempo de máquina necessário para os processos recursivos cresce como uma função quadrática com o valor da entrada a processar. É também conhecida como recursão em árvore.

```
def recfunc(parameters):
    <casos base>
    ...
    recfunc(parameters) <operador> recfunc(parameters)
    ...
```

Um bom exemplo para explicarmos a recursão binária esta na sequência de Fibonacci exemplicada a seguir

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

Esta sequência tem como característica principal que um termo , a partir do terceiro termo, é calculado somando-se os dois termos anteriores ao termo corrente. Uma forma recursiva de se representar a série de Fibonacci pode ser escrita

$$(a_n) = \begin{cases} a_0 = 1 & para\ n = 0 \\ a_1 = 1 & para\ \ n = 1 \\ a_n = a_{n-1} + a_{n-2} & para\ n > 2 \end{cases}$$

A função Fibonacci do Fragmento de Código 3.6, ilustra a recursão binária, onde pode-se observar, para a entrada da chamada da recursão, considerando n>1, é feita duas ativações da função fibonacci: a primeira com o parâmetro n-1 e a segunda com o parâmetro n-2. Neste caso dizemos que a recursão é binária.

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

Fragmento de Código 3.6. Implementação em Python da função recursiva para calcular a soma dos números da série de Fibonacci.

A Figura 3.3 ilustra parte da árvore de chamadas da função recursiva para calcular a soma dos números da série de Fibonacci com o parâmetro inicial igual a "6". Pode-se observar que fib(6) ativa as funções fib(5) e fib(4), a função fib(5) ativa as funções fib(4) e fib(3) e assim sucessivamente até que as funções de ativação sejam fib(1) ou fib(0) que retornam "1".

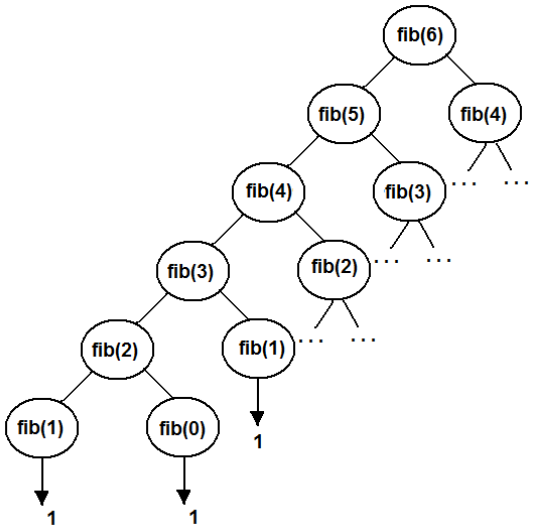


Figura 3.3 – Parte da árvore de chamadas da função recursiva para o cálculo da soma dos números da série de Fibonacci com o parâmetro inicial igual a 6.

A Figura 3.4 ilustra parte da árvore de valores de retorno das funções ativadas recursivamente na execução da função Fibonacci, para o cálculo da soma dos números da série de Fibonacci, com o parâmetro inicial igual a "6". Pode-se observar que a primeira célula desta árvore, cujo valor é "13", representa o resultado da ativação da função Fibonacci, com paramento igual a "6".

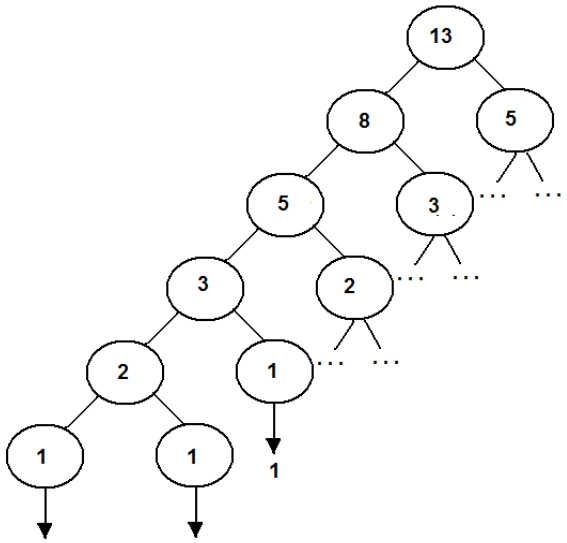


Figura 3.4 – Árvore de chamadas da função recursiva para calcular a soma dos números da série de Fibonacci, com o parâmetro inicial igual a 6, com os valores de retorno de cada função.

Otimização das funções recursivas

Nos exemplos anteriores de recursão não preocupava-se com a quantidade de vezes que a função recursiva é chamada novamente e, este problema é fundamental. Por exemplo, se na função recursiva do cálculo do fatorial do Fragmeno de Código 3.4, fosse passado como parâmetro um número muito grande, será que teríamos memória suficiente para armazer nesta **pilha** todos os parâmetros da recursão? É claro que não vamos ter memória suficiente! Nestes casos ocorre o que chamamos de estouro da pilha (Stack overflow). O que se faz para minimizar ou até mesmo eliminar este problema?

A seguir veremos a técnica de recursão denominada de recursão com chamada de cauda para minimizar o problema do estouro da pilha.

Chamada de Cauda

Neste caso, uma solução viável está em passar como parâmetro para a próxima função de recursão o resultado da função corrente. Este processo é denominado de recursão de cauda.

O Fragmento de Código 3.4 ilustra uma alternativa de eliminarmos esta **pilha** de recursividade. Na primeira vez da execução, chamamos a função *tailFatorial* e a recursão é feita pela ativação de outra função denominada de *accumulatedFatorial* que será a função de recursividade de cauda. Pode-se observar que cada função recursiva passa como parâmetro também o resultado da operação de retono da função ativadora do processo. Neste caso, a função que ativa a recursividade termina o seu ciclo, e os seus parâmetros podem ser liberados da **pilha**.

```
def accumulatedFactorial(n,f):
     if ( n == 0):
          return f
     else:
          return accumulatedFactorial(n-1, n*f)

def tailFactorial(n):
     return accumulatedFactorial(n, 1)
```

Fragmento de Código 3.4. Implementação em Python da função fatorial com recursão utilizando a chamada de cauda.

O Diagrama da Figura 3.3 ilustra a sequência de chamadas da função fatorial para o cálculo do fatorial de um número utilizando uma função recursiva de cauda. Pode-se observar que a primeira função a ser ativada é a função *tailFatorial* que tem como parâmetro “3!”. Esta função ativa a função *accumulatedFatorial* com parâmetro igual a “3” que corresponde ao número a ser calculado o seu fatorial, e o parâmetro igual a “1”, resultado parcial do valor do fatorial. A partir deste ponto começam as chamadas recursivas da função *accumulatedFatorial*, onde os parâmetros são, respectivamente, o número que queremos calcular o seu fatorial e, o resultado parcial do valor do fatorial até este ponto. Quando acontecer um determinado ponto de parada este valor é retornado para a função accumulatedFatorial que, finalmente, termina o seu ciclo e retorna o resultado do fatorial do número.

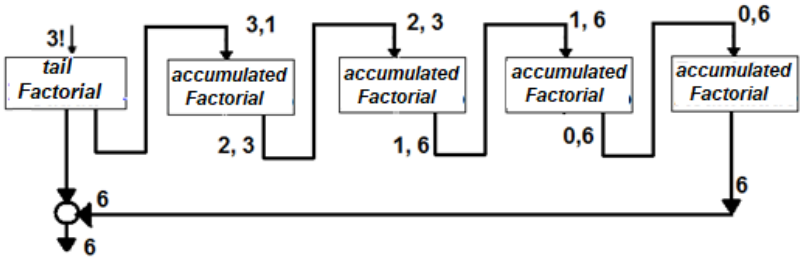


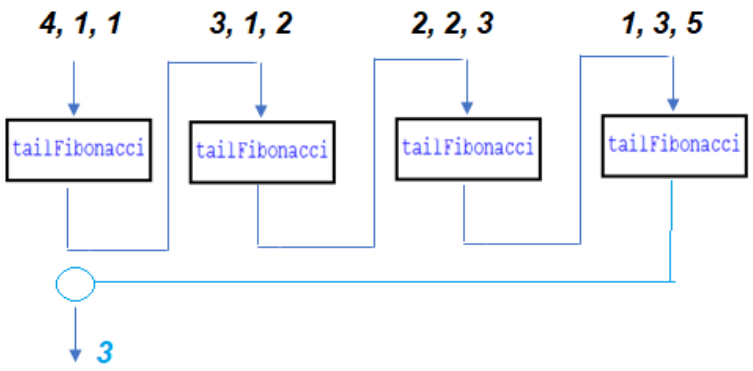
Figura 3.3 – Diagrama ilustrativo da execução da função recursiva para o cálculo do fatorial de 3 com a recursão de cauda.

As funções recursivas são geralmente consideradas mais lentas que as funções iterativas tradicionais, mas certamente simplificam muito mais as soluções de alguns problemas e, geralmente, deixam o código mais conciso e muito menor em termo de quantidade de linhas. Também é importante frisar que algumas funções matemáticas só possuem a definição recursiva. A implementação da recursão de cauda é um processo muito importante na otimização das funções recursivas, pois, além de otimizarem os seus códigos não possuem a tendência de estouro de memória no armazenamento dos parâmetros na pilha de recursividade.

A função ***tailFibonacci*** ilustrada no Fragmento de Código 3.7 representa uma adaptação da recursividade com a opção recursão com cauda. Neste caso, na primeira ativação desta função, deve-se passar como parâmetros o número de iterações, o primeiro termo da sequência, denominado de current, e o segundo termo, denominado de next. A partir daí, em cada chamada recursiva, passa-se como parâmetros, o número de iterações menos 1, pois o processo possui uma interação à menos, o número corrente da série e a soma do número corrente com o próximo (current + next).

```
def tailFibonacci(n,current,next):
     if( n == 1):
          return current
     else:
          return tailFibonacci(n-1,next,current+next)
```

Fragmento de Código 3.7. Implementação em Python da função recursiva para calcular a soma dos números da série de fibonacci utilizando a recursão de cauda.



Como foi observado na Figura 3.3, a função fibonacci sem a recursão de cauda, na sua árvore de ativação, possui uma série de repetições de chamadas, como por exemplo fib(3) é repetida 2 vezes, lembrando ainda que a árvore não está completa, pois foi representada parte dela somente. É fácil observar que teríamos ao total 3 ativações da função fib(3). Como resolver este problema?

Uma outra forma interessante de implementação da série de fibonacci é derivado da representação matricial da série de fibonacci.

Considere a matriz A quadrada definida da seguinte forma

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

Agora vamos calcular as potências

$$A^2 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$$

$$A^3 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^3 = \begin{pmatrix} 3 & 2 \\ 2 & 1 \end{pmatrix}$$

$$A^4 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^4 = \begin{pmatrix} 5 & 3 \\ 3 & 2 \end{pmatrix}$$

De forma genérica temos

$A^n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$, onde F_{n-1}, F_n e F_{n+1} são

três termos consecutivos da série de fibonacci para $n > 2$.

Calculando o determinante, em ambos os membros desta equação, obtemos a equação

$(-1)^n = F_{n+1}F_{n-1} - F_n^{\,2}$ denominada de identidade de Cassini[2]

Considerando que para toda matriz A quadrada temos: $A^n.A^m = A^{n+m}$, podemos concluir que

$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.\begin{pmatrix} F_{m+1} & F_m \\ F_m & F_{m-1} \end{pmatrix} = \begin{pmatrix} F_{n+m+1} & F_{n+n} \\ F_{n+m} & F_{n+m-1} \end{pmatrix}$ que resulta

$$F_{m+n-1} = F_m F_n + F_{m-1}F_{n-1}$$
$$F_{m+n} = F_{n+1}F_m + F_n F_{m-1}$$

Fazendo $m = n$ temos,

$F_{2n-1} = (F_n)^2 + (F_{n-1})^2$ sempre ímpar

$F_{2n} = F_{n+1}F_n + F_n F_{n-1}$ sempre par. Como $F_{n+1} = F_{n-1} + F_n$,

podemos escrever

$F_{2n} = (F_{n+1})^2 - (F_{n-1})^2$ sempre par

[2] Jean-Dominique Cassini descobriu esta fórmula em 1680

Logo, podemos escrever de forma recursiva a função de fibonacci da seguinte forma

$$(a_n) = \begin{cases} 1 \quad para \quad n = 0 \\ 1 \quad para \quad n = 1 \\ 1 \quad para \quad n = 2 \\ a_n = \left[a_{\frac{n+1}{2}} \right]^2 + \left[a_{\frac{n-1}{2}} \right]^2 \quad para \quad n \ impar \ e \ n > 2 \\ a_n = \left[a_{\frac{n}{2}+1} \right]^2 - \left[a_{\frac{n}{2}-1} \right]^2 \quad para \quad n \ par \ e \ n > 2 \end{cases}$$

O Fragmento de Código 3.8 implementa esta solução.

```
def F(n):
      if (n == 0):
            return 0 #  base case
      if (n == 1):
            return 1 #  base case
      if (n == 2):
            return 1 #  base case
      if (n % 2 != 0):
            # n is odd
            a = F((n+1)/2)
            b = F((n-1)/2)
            return a*a + b*b
      else:
            # n is even
            a = F(n/2+1)
            b = F(n/2 - 1)
            return a*a - b*b
```

Fragmento de Código 3.8. Implementação em Python da função recursiva para calcular a soma dos números da série de fibonacci utilizando a otimização baseada na identidade de Cassini.

Exercícios

Desenvolvido tomando como base os exercícios encontrado nos sítios.

https://introcs.cs.princeton.edu/Python/15inout/
https://introcs.cs.princeton.edu/Python/40algorithms/

1. **MDC com Recursão**. Escrever uma Função Recursiva para calcular o máximo divisor comum de dois números dados como parâmetros. Sabe-se que o MDC de dois números naturais não nulos **x** e **y** é dado por :

   MDC (**x**, **y**) = MDC (**x-y**, **y**) se **x** > **y**
   MDC (**x**, **y**) = MDC (**y**, **x**) se **x** < **y**
   MDC (**x, x**) = **x** se **x** = **y**

2. **Produto com Recursão**. Escreva um algoritmo iterativo para avaliar *a* * *b* usando a adição, onde *a* e *b* são inteiros não negativos e não nulos pode ser escrita da seguinte forma

$$a * b = \begin{cases} a & \text{se } b = 1 \\ a * (b-1) + a & \text{se } b > 1 \end{cases}$$

   def produto (a,b):

3. **Maior e menor elemento de um vetor e soma dos seus elementos.** Imagine **a** como um vetor de inteiros. Apresente algoritmos recursivos para calcular:

   a) O elemento máximo do vetor
   b) O elemento mínimo do vetor
   c) A soma dos elementos do vetor

4. **Potência com Recursão**. Escreva uma função recursiva que retorna o valor da potência de base b e expoente n, informados como parâmetros

   def power (b, n):

   DICA: De forma genérica:

$$b^n = \begin{cases} 1 & \text{se } n = 0 \\ 1/b^{-n} & \text{se } n < 0 \\ b * b^{n-1} & \text{se } n > 0 \end{cases}$$

5. **Soma com Recursão**. A soma de dois números inteiros positivos não nulos, pode ser definida recursivamente utilizando a idéia do sucessor e antecessor de um número.

   A função soma de dois números inteiros positivos não nulos pode ser escrita da seguinte forma:

$$soma(a,b) = \begin{cases} a & \text{se } b = 0 \\ soma(b,a) & \text{se } b > a \\ soma(sucessor(a), antecessor(b)) & \text{se } b > 1 \end{cases}$$

Por exemplo calculando a soma de 5+3 pela função teríamos

$$soma(5,3) = soma(sucessor(5), antecessor(3)) = soma(6,2) =$$
$$soma(sucessor(6), antecessor(2)) = soma(7,1) =$$
$$soma(sucessor(7), antecessor(1)) = soma(8,0) = 8$$

Como exemplo de implementação da função sucessor e antecessor, poderíamos ter:

```
def  succesor (x):
        return ( x + 1 )

def  antecesor (x):
        return ( x + 1 )
```

Escreva uma função recursiva que retorna o valor da soma de dois números inteiros positivos, informados como parâmetro, utilizando as funções sucessor e antecessor.

```
def  soma (a,b):
```

6. **Número de Zeros de um número**. Escreva uma função recursiva que retorna o número de zeros do número informado como parâmetro. Por exemplo se n= 10, a função retorna 1, se n= 10101, a função retorna 2.

```
def  zeros (x):
```

7. Dê a sequência de inteiros que serão impressos após a execução da função ex7(4):

```
def  ex7 (n):
     if n <= 0:
          return
     print (n)
     ex7(n-2)
     ex7(n-3)
     print (n)
```

8. o valor do retorno da função ex8(4):

```
def  ex8(n):
    if  n <= 0:
         return ''
    return str(ex8(n-3)) + str(n) + str(ex8(n-2)) + str(n)
```

9. Considere a função recursiva.

```
def  mystery(a,b):
     if (b == 0):
          return 0
     if (b % 2 == 0):
          return mystery(a+a, b // 2)
     return mystery(a + a, b // 2) + a
```

Considerando que o operador // significa a divisão inteira e % o resto da divisão, qual é o valor de mystery(3, 2) e mystery(4, 3)? Se escolhermos a e b inteiros quaisquer, o que representa a função mytery.

Solução. 6 e 12. A função calcula a*b

10. Considere os pares de funções a seguir que são mútuamente recursivas. Qual é o valor de `g(2)`?

```
def  f (n):
     if n == 0:
          return 0
     return n+f(n-1) + g(n-1)

def  g (n):
     if (n == 0):
          return 0
     return n+g(n-1) + f(n)
```

11. **Soma dos algarismos de um número**. Escreva uma função recursiva que retorna a soma dos algarismos de um número informado como parâmetro. Por exemplo se n= 1203, a função retorna 6, se n= 2131, a função retorna 7.

```
def digitSum(n):
```

12. **Soma dos números naturais**. Escreva uma função recursiva que retorna a soma dos n primeiros números naturais, onde n é um número informado como parâmetro. Por exemplo se n= 5, a função retorna 15, se n= 6, a função retorna 21.

```
def naturalSum( n):
```

DICA: A soma dos números naturais é a soma de uma PA de razão 1

13. **Número de Cores**. Misturando cores, a partir de um conjunto de cores iniciais, produzem-se novas cores. Supõe-se, neste método de produção, que se utilizam sempre doses iguais de cada cor misturada. Assim, partindo das cores iniciais C1 e C2 conseguem-se as seguintes 3 cores distintas: C1, C2 e C1 + C2. Com as cores iniciais C1, C2 e C3 já se conseguem 7 cores: C1, C2, C3, C1 + C2, C1 + C3, C2 + C3, e C1 + C2 + C3.

```
def nunberOfColor( n):
```

Exercícios Criativos

1. **Torres de Hanói**. Nenhuma discussão sobre a recursividade estaria completa sem o problema das antigas torres de Hanói. Temos três torres e os discos que se encaixam nos n postes. Os discos são diferentes em tamanho e são inicialmente, dispostos sobre uma das varas, a fim de o maior (disco n), na parte inferior e o menos (disco 1) na parte superior. A tarefa é mover a pilha de discos para a outra torre, obedecendo as seguintes regras:

   a) Mover apenas um disco de cada vez.
   b) Nunca coloque um disco em cima de um menor.

   Para resolver o problema, o nosso objetivo é a exibição de uma seqüência de instruções para mover os discos. Nós assumimos que os pólos estão dispostos em uma linha, e que cada instrução para mover um disco especifica seu número e se mover para a esquerda ou direita.

```
def hanoi(from, to, num):
```

2. **Inteiro para Binário**. Esta função transforma um número inteiro qualquer informado como parâmetro em binário e imprime o resultado

```
def  IntegerToBinary(n):
```

3. **Numeração de Von Neumann**. A numeração denominada de von Neumann ordinal, é uma sequência de inteiros i definida como se segue:

   para i = 0, que é o conjunto vazio {},
   para i > 0, que é o conjunto que contém os números inteiros de von Neumann 0 a i-1.

   0 = {} = {}
   1 = {0} = {{}}
   2 = {0, 1} = {{}, {{}}}
   3 = {0, 1, 2} = {{}, {{}}, {{}, {{}}}}

Escreva um programa com uma função recursiva utilizando a sequência vonNeumann que ltem com parâmetro um número N inteiro não negativo e retorna uma representação da seqüência de von Neumann para este número. Este é um método para definir ordinais na teoria dos conjuntos.

```
def vonNeumannOrdinal( n):
```

4. **Partições inteiras**. Escreva um programa que obtem um inteiro positivo N como um argumento e imprime todas as partições de N. Uma partição de N é uma maneira de escrever N como uma soma de números inteiros positivos. Duas somas são considerados o mesmo se apenas diferem na forma de seus somandos constituintes. Partições podem aparecer em polinômios simétricos e teoria da representação de grupos em matemática e física.

```
def partition(n):
```

```
Partition(3)
3
2 1
1 1 1

Partition(4)
4
3 1
2 2
2 1 1
1 1 1 1
```

5. **SubConjuntos de um Conjunto**. Escreva um programa que obtem todos os subconjuntos possíveis de um conjunto dado. Um subconjunto B do conjunto A é um conjunto tal que todos os elementos de B pertencem ao conjunto A.

```
def subSet(A):
```

```
subSet({1,2,3)
{}
{1}, {2} e {3}
{1,2}, {1,3},  {2,3}
{1,2,3}
```