Algoritmos e Funções Recursivas

Considere a definição da função fatorial:

```
n! = 1 se n ≤ 0
     n.(n-1).(n-2). . . . 3.2.1 se n > 0
```

Considere agora a seguinte definição equivalente:

```
n! = 1 se n ≤ 0
     n.(n-1)! se n > 0
```

Dizemos que essa última definição é uma definição recursiva, pois usa a função fatorial para definir a função fatorial. Ou seja, se f é a função fatorial, a definição acima fica:

```
f(n) = 1 se n ≤ 0
       n.f(n-1) se n > 0
```

A princípio parece estranho usar uma função para definir a si própria, mas vejamos como se calcula o fatorial usando a definição recursiva.

```
5!=5.4!=5.4.3!=5.4.3.2!=5.4.3.2.1!=5.4.3.2.1.0!=5.4.3.2.1.1 = 120
```

A definição acima faz sentido, pois tem um caso em que a recorrência termina. Quando n é igual a zero não há mais chamadas.

Muitas outras funções admitem definições recorrentes deste tipo, ou seja, usa-se a função para definir a si própria, com um caso particular não recorrente e a garantia que a sequência de cálculos levará a esse caso particular.

Funções recursivas em Python

Já vimos que uma função em Python pode chamar outras funções. Em particular pode chamar a si mesma. Quando isso ocorre dizemos que a função é recursiva.

A função fatorial pela definição recursiva acima ficaria:

```
def fatorial(n):
    if n <= 0: return 1
    return n * fatorial(n-1)
```

Vamos agora ilustrar as chamadas recorrentes da função fatorial.
Faremos uma pequena modificação, colocando um print() para traçarmos como as chamadas são feitas.

```
def fatorial(n):
    if n <= 0: return 1
    print("Chamando fatorial de ", n - 1)
    return n * fatorial(n - 1)

f = fatorial(5)
print("Fatorial de 5 = ", f)
```

Saida:

```
Chamando fatorial de  4
Chamando fatorial de  3
Chamando fatorial de  2
Chamando fatorial de  1
Chamando fatorial de  0
Fatorial de 5 =  120
```

Números Harmônicos

Considere agora a função que calcula o `n`-ésimo número harmônico:

```
H(n) = 1 + 1/2 + 1/3 + 1/4 + . . . + 1/n (n >= 1)
```

Uma definição recursiva:

```
H(n) = 1 se n ≤ 1
       1/n + H(n-1) se n > 1
```

Usando a definição recursiva acima:

```
H(4)=1/4+H(3)=1/4+1/3+H(2)=1/4+1/3+1/2+H(1)=1/4+1/3+1/2+1
```

Análogo ao fatorial, a função acima também tem o caso de parada (`n` igual a `1`), onde o valor da função não é recorrente.

Portanto, usando a definição acima:

```
def H(n):
    if n <= 1: return 1
    return 1 / n + H(n - 1)
```

Soma dos elementos de uma lista

Considere uma lista de números, por exemplo [23, 45, 87,12] dos quais queremos a soma. Já sabemos como percorrer a lista, acumulando a soma dos elementos. Há outra forma de pensar na soma dos elementos de uma lista:

Soma da lista = 0 (se a lista está vazia)
primeiro elemento + Soma do resto da lista (se a lista não está vazia)

```
def soma_da_lista(lista):
    if lista == []: return 0
    return lista[0] + soma_da_lista(lista[1:])
```

Exercícios: Para cada um dos exercícios abaixo, escreva a função de 2 formas: iterativa e recursiva.

P4.1) Função Max_Lista(L) que devolve o máximo dos elementos de uma lista L.

```
def Maior(x, y):
    if x > y: return x
    else: return y

def Max_Lista(L):
    if len(L) == 1: return L[0]
    return Maior(L[0], Max_Lista(L[1:]))
```

P4.2) Idem Min_Lista(L) para o mínimo.

P4.3) Função Inverte_Lista(L) que devolve a lista L invertida.

P4.4) Função Potencia(x, n) que recebe $x \neq 0$ float e $n \geq 0$ inteiro e devolve x^n

P4.5) Função Soma_Seq(x, n) que recebe x float e $n \geq 0$ inteiro e devolve o valor da soma:

$x + x^2/2 + x^3/3 + ... x^n/n$

P4.6) Função Soma_Seq(x, eps) que recebe x float e eps float (eps - valor pequeno – ex: 10^{-5} x é um valor entre 0 e 1) e devolve o valor da soma acima até encontrar um termo que seja menor que eps.

P4.7) Função Busca(L, x) que recebe uma lista L e um valor x e procura x em L. Devolve o primeiro k tal que L[k] = x ou -1 se não encontrar. Está resolvido à frente, mas tente resolvê-lo sem ver a solução.

P4.8) Função Binomial(n, k) que devolve o número binomial de n e k. A definição recursiva do número binomial de n e k é:

0 se se $n = 0$ e $k > 0$
1 se se $n \geq 0$ e $k = 0$
Binomial(n-1, k) + Binomial(n-1, k-1) se n e $k > 0$

Comentários sobre as soluções recursivas

Nos casos acima, não parece existir nenhuma vantagem da solução recursiva para a solução iterativa. De fato, a solução recursiva é até pior em termos de gasto de memória e de tempo, pois a cada chamada é necessário guardar o contexto da chamada anterior, até que ocorra o caso particular não recorrente.

Entretanto, a estrutura da função recursiva fica análoga à sua definição. Isso já é uma grande vantagem.

A recursão é um método de solução que procura expressar a solução de um problema em termos de soluções mais simples do mesmo problema, até chegar a um ponto que o mesmo é resolvido de maneira trivial.

Entretanto em alguns casos, esse método de solução, nos permite escrever soluções que, de outra forma, seriam muito difíceis de programar. Veremos alguns casos abaixo.

Uma solução recursiva deve sempre possuir as seguintes características:

- O algoritmo recursivo deve possuir um **caso base**, ou um caso em que não seja necessária a chamada de si mesmo;
- O algoritmo recursivo deve alterar o seu estado de maneira a **se aproximar do caso base**;
- O algoritmo recursivo deve ter uma **chamada a si mesmo** (direta ou indiretamente), ou seja, dentre os comandos, inclusive chamadas de funções que o compõe deve haver uma chamada de si mesmo.

Nos exemplos vistos acima, a chamada recursiva substitui a repetição, usada nos comandos while e for.

A sequência de Fibonacci

A sequência de Fibonacci, assim conhecida porque foi proposta pelo matemático italiano do século XI, Leonardo Fibonacci, é tal que cada elemento (com exceção dos dois primeiros que são `0` e `1`), é igual à soma dos dois anteriores.

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

A sequência possui algumas propriedades matemáticas que não são objeto de análise neste curso.

$f(0) = 0$
$f(1) = 1$
$f(n) = f(n-1) + f(n-2)$

P4.9) Função Fibonacci(n). Recebe n > 1 e devolve o n-ésimo número da sequência de Fibonacci.

Solução iterativa:

```
def Fibonacci(n):
    if n == 0: return 0  # casos
    if n == 1: return 1  # particulares
    anterior, atual = 0, 1
    # repetir até o n-ésimo inclusive
    for k in range(2, n + 1, 1):
        anterior, atual = atual, anterior + atual
    return atual
```

Solução recursiva:

```
def Fibonacci(n):
    if n == 0: return 0  # casos
    if n == 1: return 1  # particulares
    return Fibonacci(n - 1) + Fibonacci (n - 2)
```

Observe que a versão recursiva é mais elegante que a iterativa. Entretanto é menos eficiente. Para se calcular Fibonacci(3) por exemplo, na versão recursiva, serão geradas 4 chamadas.

P4.10) Quantas chamadas são necessárias, para se calcular Fibonacci(n) $n \geq 0$?

Máximo Divisor Comum – Algoritmo de Euclides

Um exemplo que converge rapidamente para o caso trivial, é o cálculo do `mdc` entre dois números, usando o algoritmo de Euclides.

Quociente		1	1	2
dividendo/divisor	30	18	12	6
Resto	12	6	0	

Veja a versão iterativa:

```
def mdc(n, m):
    dividendo, divisor, resto = n, m, n % m
    while resto != 0:
        dividendo, divisor, resto = divisor, resto, divisor % resto
    return divisor
```

Vamos usar a seguinte definição (recursiva) para o mdc:

```
mdc(a, b) = b se b divide a, ou seja a % b é 0
            mdc(b, a % b) caso contrário
```

Portanto a versão recursiva ficaria:

```
def mdc(n, m):
    if n % m == 0: return m
    return mdc(m, n % m)
```

Busca Sequencial

Considere a função de busca sequencial em tabela:

```
# Procura x na lista a de n elementos.
# Devolve o índice do primeiro encontrado ou -1 se não encontrar
def busca(a, n, x):
    for k in range(n):
        if a[k] == x: return k
    return -1
```

Vamos explorar agora soluções recursivas para a busca sequencial.

Busca Sequencial recursiva – introduzindo mais um parâmetro

Uma definição recursiva da função busca:

Se o primeiro elemento for igual a x, encontrou e devolve o índice deste elemento, senão repita o processo para o restante da tabela. Para essa solução teremos que ter como parâmetro também, o índice do primeiro elemento do restante da tabela. A chamada inicial da função seria então busca(a, n, x, 0) e a definição fica:

```
busca (a, n, x, k) =     -1 se k = n
                         k se a[k] = x
                         busca(a, n, x, k+1) se a[k] ≠ x
```

```
# Procura x na lista a de n elementos.
# Devolve o índice do primeiro encontrado ou -1 se não encontrar
def busca(a, n, x, k):
    if k == n: return -1
    if a[k] == x: return k
    return busca(a, n, x, k+1)

# exemplo de chamadas da função busca

v = [1, 2, 3, 4]
print(busca(v, 4, 3, 0))
print(busca(v, 4, 5, 0))
```

Busca Sequencial recursiva – sem introduzir novo parâmetro

Podemos usar o próprio n (número de elementos da tabela) como contador. Porém, nesse caso a tabela seria varrida do fim para o começo e o índice devolvido seria o do último elemento igual a x:

```
busca (a, n, x) =   -1 se n = 0
                    n-1 se a[n-1] = x (último elemento da tabela)
                    busca(a, n-1, x) se a[k] ≠ x
```

```
# Procura x na lista a de n elementos.
# Devolve o índice do último elemento igual a x ou -1 se não
encontrar
def busca(a, n, x):
    if n == 0: return -1
    if a[n - 1] == x: return n - 1
    return busca(a, n - 1, x)

# exemplo de chamadas da função busca

v = [1, 2, 3, 4]
print(busca(v, 4, 3))
print(busca(v, 4, 5))
```

Apenas lembrando, `if a[n-1] == x:`, é o mesmo que `if a[-1] == x:`.

Busca Sequencial recursiva – devolvendo apenas True ou False

Se não for importante devolver o índice do elemento igual a x e apenas se encontrou ou não (True ou False) podemos usar sub-listas para resolver recursivamente. De fato, nem precisamos do parâmetro n:

```
busca (a, x) = False se lista vazia
               True se a[0] = x (primeiro elemento da lista)
               busca(a[1: em diante], x) se a[0] ≠ x
```

```
# Procura x na lista a.
# Devolve True ou False - usando sub-listas
def busca(a, x):
    if len(a) == 0: return False
    if a[0] == x: return True
    return busca(a[1:], x)
```

A solução acima percorre a lista do início. O mesmo pode ser feito percorrendo a partir do final.

P4.11) Idem a versão anterior percorrendo a lista a partir do final da mesma.

P4.12) Idem versão anterior devolvendo o índice do primeiro elemento encontrado. Sugestão: em vez `return busca(a[1:], x)`, podemos fazer `return 1 + busca(a[1:], x)`. Modifique a definição para devolver algo diferente de -1 se não encontrar.

P4.13) Repita as soluções recursivas anteriores para a função busca, eliminado o parâmetro n (use a função len e sub-listas).

Busca Binária

Já vimos a versão iterativa da busca binária. Esse algoritmo admite claramente uma versão recursiva:

Busca:

Compare com o elemento do meio da tabela. Se for igual achou.
Se for maior faça a **Busca** na parte inferior da tabela.
Se for menor faça a **Busca** na parte superior da tabela.

No algoritmo acima também devemos parar quando a tabela está vazia (zero elementos).

Vamos fazer a função de modo que a chamada inicial tem como parâmetros os índices inicial e final da tabela, zero e n-1:

```
k = Busca_Binaria(a, x, 0, n-1)
```

```
def Busca_Binaria(L, x, inicio, fim):
    if inicio > fim: return -1 # tabela vazia
    meio = (inicio + fim) // 2
    if x == L[meio]: return meio
    if x > L[meio]: return Busca_Binaria(L, x, meio + 1, fim)
    return Busca_Binaria(L, x, inicio, meio - 1)
```

P4.14) Reescreva a função acima apenas com o parâmetros L e x (`def Busca_Binaria(L, x)`). Use o tamanho de L e sub-listas.

Valor de polinômio - Regra de Horner

Dados `x, a[n], a[n-1], ..., a[0]` calcular:
$P(n, x) = a[n]x^n + a[n-1]x^{n-1} + \ldots + a[1]x + a[0]$.

Para, para $n = 4$, o cálculo:
$a[4]x^4 + a[3]x^3 + a[2]x^2 + a[1]x + a[0]$ que pode ser escrito pela regra de Horner como:

$(((a[4]x + a[3])x + a[2])x + a[1])x + a[0]$

Com quatro multiplicações e quatro somas apenas calculamos o valor. No caso geral, com apenas n multiplicações e n somas o valor do polinômio no ponto x é calculado.

Os coeficientes estão numa lista de n + 1 elementos. Abaixo uma definição iterativa:

```
def P (a, x, n):
    soma = a[n]
    k = n - 1
    while k >= 0:
        soma = soma * x + a[k]
        k = k - 1
    return soma
```

Observe agora que:

$a[n]x^n + \ldots + a[0] = x(a[n]x^{n-1} + \ldots + a[1]) + a[0]$ **se n > 0**
$a[0]$ **se n = 0**

P12.15) Escreva uma função recursiva para resolver o problema acima

Potência com expoente inteiro

Embora já exista a função intrínseca **`pow(x, y)`** para calcular x^y, para x e y genéricos, vamos tentar achar soluções melhores para o caso de expoente inteiro maior ou igual a zero.

Seja $f(x, n) = x^n$ ($n \geq 0$ inteiro).
Observe que $f(x, n) = 1$ se $n = 0$ e $x * f(x, n - 1)$ se $n > 0$

P12.16) Escreva a função acima nas versões iterativa e recursiva.

A solução recursiva acima não apresenta vantagens, pois continuamos fazendo n multiplicações.

Será possível diminuir a quantidade de multiplicações?

Em se tratando de multiplicações de números não faz muito sentido diminuir pois é uma operação muito simples. Suponha por exemplo que x é uma matriz de n x n elementos em vez de um número. Neste caso, faz todo sentido procurar algoritmos que minimizem o número de multiplicações.
Veja que:

$x^4 = x^2.x^2$ e $x^2 = x.x$ (x^4 com 2 multiplicações)
$x^5 = x^4.x$, $x^4 = x^2.x^2$ e $x^2 = x.x$ (x^5 com 3 multiplicações)
$x^{10} = x^5.x^5$, $x^5 = x^4.x$, $x^4 = x^2.x^2$ e $x^2 = x.x$ (x^{10} com 4 multiplicações)

Podemos então fazer menos multiplicações usando produtos anteriores.

Veja a seguinte definição:

$$x^n = \begin{cases} 1 \text{ se } n = 0;\ x \text{ se } n = 1 \\ (x^h)^2 \text{ se } n > 1 \text{ e par} \\ (x^h)^2.x \text{ se } n > 1 \text{ e ímpar} \\ \text{Onde h é o quociente de n/2 truncado.} \end{cases}$$

P12.17) Faça uma função pot(x, n) recursiva usando a definição acima. Verifique na sua solução quantas chamadas à função pot são feitas. O objetivo é que a função faça o mínimo de chamadas possível.

P12.18) Quantas multiplicações são feitas para se calcular x^n no algoritmo acima ?

P12.19) Refaça os 2 exercícios anteriores expandindo a definição para n negativo, lembrando que:
$x^n = 1 / x^{-n}$ se $n < 0$. Se $n < 0$ e $x = 0$ a função não está definida.

P12.20) Dado n e uma sequência com n números, imprimir a sequência na ordem inversa a que foi lida. Fazer isso sem usar listas. Sugestão: faça uma função recursiva `imprime`, que lê um número, chama a si própria se não chegou ao fim da sequência e imprime o número lido.

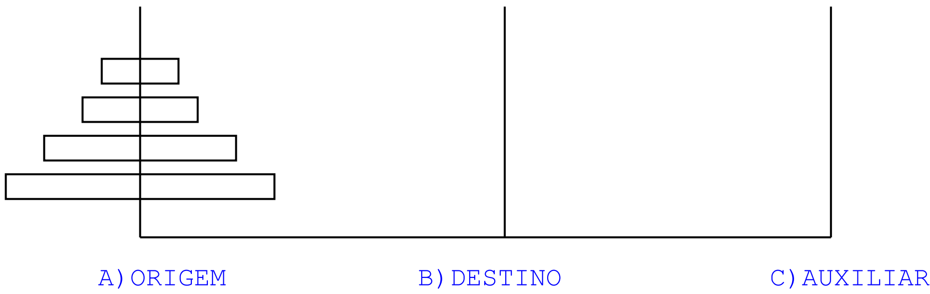
P12.21) Idem para uma sequência terminada por zero.

Torre de Hanói

Em todos os exemplos acima, a versão recursiva, embora às vezes mais elegante, pois é mais aderente a definição da função, não é mais eficiente que a versão iterativa. Em alguns casos, no entanto, pensar na solução de forma recursiva, facilita muito.

O jogo Torre de Hanói, foi inventado pelo matemático francês Édouard Lucas (1842-1891), é um exemplo de problema que tem uma solução simples na forma recursiva.
O problema consiste em mover n discos empilhados (os menores sobre os maiores), de uma haste de origem, para uma haste de destino, na mesma ordem, respeitando as seguintes regras:

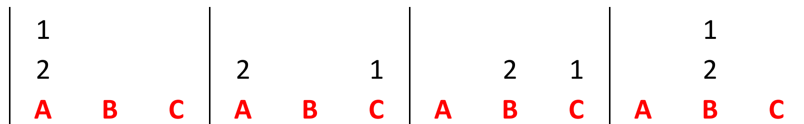
1) Apenas um disco pode ser movido por vez
2) Não colocar um disco maior sobre um menor
3) Pode usar uma haste auxiliar



Por exemplo, uma solução para 2 discos seria:

```
move disco  1 da torre    ORIGEM para a torre   AUXILIAR
move disco  2 da torre    ORIGEM para a torre    DESTINO
move disco  1 da torre  AUXILIAR para a torre    DESTINO
```

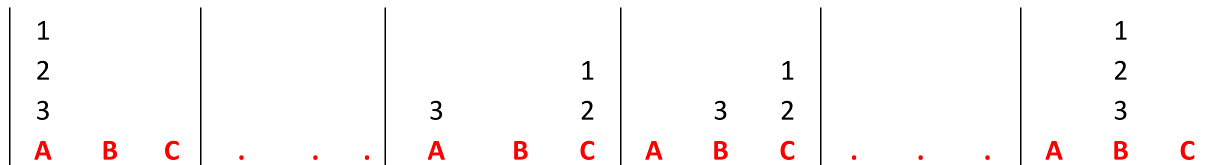
Correspondente aos movimentos abaixo:



E para 3 discos, uma solução seria:

```
move disco  1 da torre    ORIGEM para a torre    DESTINO
move disco  2 da torre    ORIGEM para a torre   AUXILIAR
move disco  1 da torre   DESTINO para a torre   AUXILIAR
move disco  3 da torre    ORIGEM para a torre    DESTINO
move disco  1 da torre  AUXILIAR para a torre     ORIGEM
move disco  2 da torre  AUXILIAR para a torre    DESTINO
move disco  1 da torre    ORIGEM para a torre    DESTINO
```

Correspondem aos movimentos abaixo:



Tente fazer para 4 e 5 discos e veja o enorme trabalho que dá.

Vejamos como seria uma função recursiva para resolver este problema.
Supondo que saibamos como mover (n – 1) discos respeitando as regras acima. Vamos então mover n discos:

Mover n discos de ORIGEM para DESTINO, é o mesmo que:

- Mover n-1 discos de ORIGEM para AUXILIAR, usando DESTINO como auxiliar

- Mover disco n de ORIGEM para DESTINO
- Mover n-1 discos de AUXILIAR para DESTINO, usando ORIGEM como auxiliar

Correspondem aos movimentos abaixo:

1																1	
2								1			1					2	
3								2			2					3	
…								3			3					…	
n-1								…			…					n-1	
n						n		n-1		n	n-1					n	
A	B	C	.	.	.	A	B	C	A	B	C	.	.	.	A	B	C

Em qualquer das operações acima, não transgredimos as regras do jogo. Falta a regra de parada que ocorre quando n é 1. Neste caso, basta mover o disco 1 de ORIGEM para DESTINO.

Usando a definição acima a função ficaria:

```
def Hanoi(n, torreA, torreB, torreAux):
    if n == 1:
        # mover disco 1 da torreA para a torreB
        Movimente(1, torreA, torreB)
    else:
        # n - 1 discos da torreA para torreAux com torreB auxiliar
        Hanoi(n - 1, torreA, torreAux, torreB)
        # mover disco n da torreA para torreB
        Movimente(n, torreA, torreB)
        # n - 1 discos da torreAux para a torreB com torreA auxiliar
        Hanoi(n - 1, torreAux, torreB, torreA)

def Movimente(k, origem, destino):
    print("mover disco ", k, " da torre ", origem, " para a torre ", destino)
```

Veja as chamadas abaixo:

```
print("\n\n2 discos de A para B usando C")
Hanoi(2, "A", "B", "C")
print("\n\n3 discos de Amarela para Branca usando Azul")
Hanoi(3, "Amarela", "Branca", "Azul")
print("\n\n4 discos de 1 para 2 usando 3")
Hanoi(4, 1, 2, 3)
```

Saida:

```
2 discos de A para B usando C
mover disco  1  da torre  A  para a torre  C
mover disco  2  da torre  A  para a torre  B
mover disco  1  da torre  C  para a torre  B


3 discos de Amarela para Branca usando Azul
```

```
mover disco  1  da torre  Amarela  para a torre  Branca
mover disco  2  da torre  Amarela  para a torre  Azul
mover disco  1  da torre  Branca  para a torre  Azul
mover disco  3  da torre  Amarela  para a torre  Branca
mover disco  1  da torre  Azul  para a torre  Amarela
mover disco  2  da torre  Azul  para a torre  Branca
mover disco  1  da torre  Amarela  para a torre  Branca


4 discos de 1 para 2 usando 3
mover disco  1  da torre  1  para a torre  3
mover disco  2  da torre  1  para a torre  2
mover disco  1  da torre  3  para a torre  2
mover disco  3  da torre  1  para a torre  3
mover disco  1  da torre  2  para a torre  1
mover disco  2  da torre  2  para a torre  3
mover disco  1  da torre  1  para a torre  3
mover disco  4  da torre  1  para a torre  2
mover disco  1  da torre  3  para a torre  2
mover disco  2  da torre  3  para a torre  1
mover disco  1  da torre  2  para a torre  1
mover disco  3  da torre  3  para a torre  2
mover disco  1  da torre  1  para a torre  3
mover disco  2  da torre  1  para a torre  2
mover disco  1  da torre  3  para a torre  2
```

P4.22) Mostre que para N discos, o algoritmo acima precisa de 2^N-1 movimentos.

Sudoku

O jogo Sudoku consiste em preencher os espaços vazios de uma matriz 9x9 com os algarismos de 1 a 9 de tal maneira que em cada linha, cada coluna e cada quadrado interno de 3x3 contenha todos os algarismos sem repetição. Alguns dos espaços já estão preenchidos.
O jogo possui graduações de dificuldade. Normalmente é publicado como Muito Fácil, Fácil, Médio, Difícil e Muito Difícil. Essa graduação está associada à quantidade de possibilidades para se chegar à solução. Quanto mais possibilidades, mais difícil. Em geral, quanto menos espaços já preenchidos são fornecidos, mais difícil será o preenchimento. Exemplos de matrizes Sudoku:

				1	9			6
			3		6			5
					4		2	9
4		5	1			9		
	3						5	
		1			5	2		8
2	1		9					
6			7		1			
5			6	2				

		4		6			9	
	9			2	7			
3			5	9		2		1
					6	5		
8		5				3		9
		2	9					
7		3		1	4			5
			7	5			3	
	4			3		8		

Se quiser exercitar suas habilidades no Sudoku, visite o site:
https://www.geniol.com.br/logica/sudoku

Para cada posição vazia, pode haver mais de um candidato. Para resolver o problema então basta colocar os candidatos em cada uma das posições até encontrar uma combinação de candidatos que preencha toda a matriz. Pode haver mais de uma solução ou eventualmente nenhuma.

Na solução convencional vamos experimentando o preenchimento de cada posição por tentativa e erro. O mesmo princípio pode ser usado num algoritmo.
A solução convencional pode ser expressa da seguinte forma:

Preencher a matriz S:

1. Procure uma casa livre – seja S[lin][col] essa posição
   Se não há posição livre já temos uma solução – Mostre a solução e retorne dessa chamada
   Senão, para cada um dos candidatos a essa posição:
   a. Faça S[lin][col] = candidato
   b. Prencher a matriz S (agora com mais uma posição preenchida)
2. Faça S[lin][col] = 0 (ou seja, considere essa posição livre novamente)
   não há mais candidatos que atendam a essa posição
   retorne dessa chamada

Veja que estamos invocando o preenchimento dentro do preenchimento, mas com pelo menos uma posição a mais já preenchida.
Esse tipo de solução é denominada de back-tracking. Vamos preenchendo as casas vazias mas mantemos a trilha das possibilidades de preenchimento das casas anteriores.

Detalhando um pouco mais o algoritmo, vamos a partir da posição S[0][0]:

```
# Chamada inicial da função
PreencheSudoku(S, 0, 0)

# Preenche a matriz Sudoku
def PreencheSudoku(S, L, C):
     Procure uma posição livre a partir de S[L][C]
     Se não há:
          # A matriz S está completa.
          Mostre S.
          retorne desta chamada - eventualmente há outras soluções
     Seja S[lin][col] a nova posição livre.
     # Podem haver zero ou mais candidatos a ocupar essa posição.
     # Os candidatos não podem estar na linha lin, na coluna col
     # ou no quadrado correspondente.
     Para cada candidato:
          S[lin][col] = candidato.
          PreencheSudoku(S, lin, col)
     # Neste ponto não há mais candidatos para a posição S[lin][col]
     # Considere essa posição livre e retorne desta chamada para que
     # chamadas anteriores da função possam tentar o preenchimento
     S[lin][col] = 0
     retorne desta chamada
```

P4.23) Complete a função acima resolvendo o problema do Sudoku.