13 Aula 13: 24/SET/2019

13.1 Aulas passadas

Nas aulas passadas tratamos de recursão usando os problemas:

- torres de Hanoi (aula 11): mostrou como o raciocínio recursivo auxilia na solução de problemas;
- `fatorialR()` e `fatorialI()` (aula 11): ilustrou o mecanismo recursivo através de um exemplo simples;
- `hanoi()` (aula 12): análise de algoritmos, relações de recorrência, solução exponencial;
- `fibonacciR()` e `fibonacciI()`(aula 12): mostrou uso inapropriado de recursão resolvendo um mesmo subproblema várias várias várias vezes; valores calculados deveria ser guardados/memorizados; voltaremos a considerar isso mais para frente;
- `maximoR()` (aula 12): exercitou raciocínio recursivo; evidência a diferença entre fatiamento (são clones) e vistas (são apelidos).
- `binomialR()` e `binomialI()`: conceitualmente identico a `fibonacciR()` e `fibonacciI()`;

Um problema típico de soluções recursivas é o recálculo de valores, tornando o processo muuuiiiitooo lento. Mais para frente vamos introduzir o conceito de `memoization` utilizado para evitar o recálculo de valores e deixar a computação mais eficiênte. Para isso podemos discutir os seguintes problemas:

Falamos das **três leis de recursão** que um algoritmo recursivo deve obedecer, grosseiramente:

- ter um **caso base**;
- alterar seu estado de maneira a se **aproximar do caso base**;
- chamar a si mesmo direta ou indiretamente.

13.2 Aula de hoje

Resolvemos o problema do labirinto: *busca em profundidade.*

Além disso:

- `maximoR()`: fatias versus vistas, ou
- `mdc()`: algoritmo de Euclides.

13.3 Labirinto

Arquivos:

- `caminho_animacao.py`: recebe o nome de um arquivo com labirinto. Para começar a executar precisa teclar `ENTER` no console que iniciou o programa. Os arquivos com labirintos tem extensão `.txt` (`lab.txt`, `beco.txt`, `lab2.txt`, `beco2.txt`).
- `caminho.py`: é uma versão ASCII de animação. Tem um labirinto *hardcoded*. ajuda para fazer um passo a passo enquanto a função está sendo desenvolve.

Problema:

Dada uma posição em um labirinto, encontrar um caminho até a saída. Cada posição do labirinto é formada por um quadrado que pode estar vazio ou conter uma parede.

13.3.1 Representação do labirinto

Um labirinto pode ser representado, essencialmente, por uma lista de listas de caracteres. Uma posição com um `'+'` indica uma parede e uma posição vazia é representada por um `' '`. Um `'I'` pode indicar a posição inicial. Para facilitar o código `'X'` é a posição da saída.

Vamos supor o labirinto todo murado, exceto pela saída que contém um `'X'`.

```
CAMINHO = 'C'
MARCA   = '*'
PAREDE  = '+'
BECO    = 'b'
SAIDA   = 'X'
ORIGEM  = 'I'

def main():
    lab = [
        ['+','+','+','+','+','+','+','+','+','+','+'],
        ['+',' ',' ',' ',' ',' ',' ','+',' ',' ','X'],
        ['+',' ','+',' ','+','+',' ','+',' ','+','+'],
        ['+',' ','+',' ',' ',' ',' ','+',' ','+','+'],
        ['+','+','+',' ','+','+',' ','+',' ',' ','+'],
        ['+',' ',' ',' ','+','+',' ',' ',' ',' ','+'],
        ['+','+','+',' ','+','+','+','+','+',' ','+'],
        ['+',' ',' ',' ','+','+',' ',' ','+',' ','+'],
        ['+',' ','+','+',' ',' ','+',' ',' ',' ','+'],
        ['+',' ',' ',' ',' ',' ','+',' ','+','+','+'],
        ['+','+','+','+','+','+','+','+','+','+','+']
    ]
    lin_ini, col_ini = 9, 5
    lab[lin_ini][col_ini] = ORIGEM
    procure_saida(lab, lin_ini, col_ini)
```

13.3.2 Solução

```python
#------------------------------------------------------------
def procure_saida(lab, lin, col):
    '''(list, int, int) -> bool

    Recebe uma matriz `lab` de strings representando um labirinto e
    uma posição [lin][col] para onde desejamos nos mover.

    Os símbolos encontrados em cada posição de `lab` são

        SAIDA   = 'X'
        MARCA   = '*'
        PAREDE  = '+'
        BECO    = '-'
        VAZIA   = ' '
        CAMINHO = 'C'
        ORIGEM  = 'M'
    '''
    #--------------------------------------------------------------
    # BASE
    # 1. encontramos a saída
    if lab[lin][col] == SAIDA:
        lab[lin][col] = CAMINHO
        return True

    # 2. 'entramos' em uma parede
    if lab[lin][col] == PAREDE:
        return False

    # 3. chegamos em um posição que já foi examinada
    if lab[lin][col] == TENTOU or lab[lin][col] == BECO:
        return False

    lab[lin][col] = TENTOU

    #----------------------------------------------------
    # tente cada uma das quatro direções a partir da [lin][col],
    # se necessário: NORTE, OESTE, SUL, LESTE
    encontrou = procure_saida(lab, lin-1, col) or \
                procure_saida(lab, lin,   col-1) or \
                procure_saida(lab, lin+1, col) or \
                procure_saida(lab, lin,   col+1)

    if encontrou:
        lab[lin][col] = CAMINHO
    else:
        lab[lin][col] =  BECO
    return encontrou
```

13.4 Máximo

Slide: slides_maximo.pdf

Programas: maximoR0.py, maximoR1.py, maximoR2.py, maximoR3.py

O objetivo aqui é exercitar o raciocínio recursivo e mostrar que fatiamento não é de graça.

13.4.1 maximoI()

```
def maximoI(v):
    '''(list) -> item

    Recebe um inteiro positivo n e uma lista v e retorna
    o maior elemento das n primeiras posições.

    Consumo de tempo linear.
    '''
    n = len(v)
    maxi = v[0]
    for i in range(1, n):
        if maxi < v[i]:
            maxi = v[i]
    return maxi
```

13.4.2 maximoR()

```
def maximo(v):
    return maximoR(v, len(v))

def maximoR(v, n):
    '''(list, int) -> item

    Recebe um inteiro positivo n e uma lista v e retorna
    o maior elemento das n primeiras posições.

    Consumo de tempo linear.
    '''
    if n == 1: return v[0]
    x = maximoR(v, n-1)
    if x > v[n-1]: return x
    return v[n-1]
```

13.4.3 maximoR2()

```
def maximoR2(v):
    '''(list ou array) -> item
```

```
Recebe um inteiro positivo n e uma lista v e retorna
o maior elemento das n primeiras posições.

Consumo de tempo quadrático para lista e linear para fatia.
'''
n = len(v)
if n == 1: return v[0]
x = maximoR2(v[:n-1])
if x > v[n-1]: return x
return v[n-1]
```

13.5 MDC

O **máximo divisor comum** de dois números é o maior número positivo que divide ambos.

13.5.1 solução ingênua

```
#-----------------------------------------------
def mdc(m, n):
    '''(int,int) -> int

    Recebe inteiros positivos m e n e retorna
    o máximo divisor comum de m e n.
    '''

    # condidato a divisor
    d =  min(m,n)

    while m % d != 0 or n % d != 0:
        d -= 1

    return d
```

13.5.2 Algoritmo de Euclides

```
#--------------------------------------------
def euclidesI(m, n):
    '''(int, int) -> int

    Recebe inteiros não negativos m e n e retorna
    o seu máximo divisor comum.

    Pré-condição: a função supões que n > 0.
    '''
    r = m % n
    while r != 0:
        m = n
        n = r
        r = m % n

    return n

#----------------------------------------------------------
def euclidesR(m, n):
    '''(int,int) -> int

    Recebe inteiros m e n e retorna o máximo divisor
    comum de m e n.
```

```
    Pré-condição: a função supões que m > 0.
    '''

    if n == 0: return m
    return euclidesR (n, m % n, profundidade + 1)


#----------------------------------------------------------
def euclidesR_s(m, n, profundidade):
    '''(int,int,int) -> int

    Recebe inteiros m e n e retorna o máximo divisor
    comum de m e n.
    A função também recebe um inteiro profundidade
    que indica a profundidade da recursão e é usado
    para ilustrar as chamadas recursivas.

    Pré-condição: a função supõe que m > 0.

    Exemplo:
    >>> euclidesR_s(16,42,0)
    euclidesR(16,42)
      euclidesR(42,16)
        euclidesR(16,10)
          euclidesR(10,6)
            euclidesR(6,4)
              euclidesR(4,2)
                euclidesR(2,0)
    2
    >>>
    '''
    print("  "*profundidade, end="")
    print("euclidesR(%d,%d)" %(m, n))

    if n == 0: return m
    return euclidesR_s (n, m % n, profundidade + 1)
```