

**MAC2166 – Introdução à Computação**  
ESCOLA POLITÉCNICA  
Prova Substitutiva – 8 de julho de 2014

Nome: \_\_\_\_\_

Assinatura: \_\_\_\_\_

Nº USP: \_\_\_\_\_ Turma: \_\_\_\_\_

Professor: \_\_\_\_\_

**Instruções:**

1. Não destaque as folhas deste caderno.
2. A prova consta de 3 questões. Verifique antes de começar a prova se o seu caderno de questões está completo.
3. A prova pode ser feita a lápis. Cuidado com a legibilidade e, principalmente, com a TABULAÇÃO.
4. Qualquer questão pode ser resolvida em qualquer página. Se a questão não está na página correspondente ao enunciado basta indicar isto na página e escrever QUESTÃO X em letras ENORMES antes da solução.
5. Não é necessário apagar rascunhos no caderno de questões.
6. Não é permitido o uso de folhas avulsas para rascunho.
7. Não é permitido o uso de equipamentos eletrônicos.
8. Não é permitido a consulta a livros, apontamentos ou colegas.
9. Você pode definir funções e usá-las à vontade.

**DURAÇÃO DA PROVA: 2 horas**

Questão	Valor	Nota
1	2,0	
2	4,0	
3	4,0	
Total	10,0	

## QUESTÃO 1 (2,0 pontos)

Escreva uma função

```
def remova(seq, subseq):  
    ''' (lista,lista) -> lista '''
```

que recebe duas listas `seq` e `subseq` de números inteiros. A função deve **criar** e **retornar** uma lista nova que é uma cópia de `seq` porém apagando-se a primeira ocorrência dos elementos de `subseq` em `seq`. A remoção deve respeitar a ordem de ocorrência dos elementos em ambas as listas da esquerda para a direita.

### Exemplos:

```
Para          seq      = [1, 2]  
              subseq   = [1, 1, 2, 2, 1, 2]  
a função deve retornar nova = [2]
```

```
Para          seq      = [1, -1, 3, 2, -10, -1, -12, -1, -2, -1, 8, 1, 7, 3, 5]  
              subseq   = [1, 3, -1, -1]  
a função deve retornar nova = [-1, 2, -10, -12, -2, -1, 8, 1, 7, 3, 5]
```

```
Para          seq      = [1, 1, 2, 2, 1, 1, 2, 2, 1, 1, 2, 2, 1, 1, 2, 2]  
              subseq   = [1, 2]  
a função deve retornar nova = [1, 2, 1, 1, 2, 2, 1, 1, 2, 2, 1, 1, 2, 2]
```

```
Para          seq      = [15, 15, 15, 15, 15, 15, 15, 15]  
              subseq   = [15, 15, 15, 15, 15, 15, 15]  
a função deve retornar nova = [15]
```

```
Para          seq      = [5, -1, 6, 7]  
              subseq   = [5, -1, 6, 7, 10, 8, 17]  
a função deve retornar nova = []
```

#-----

# Solução

```
def remova(seq, subseq):  
    n_seq      = len(seq)  
    n_subseq   = len(subseq)  
  
    nova = [] # cria uma lista  
    j = 0  
    for i in range(n_seq):  
        if j < n_subseq and seq[i] == subseq[j]:  
            j = j + 1  
        else:  
            nova.append(seq[i])  
  
    return nova
```

## QUESTÃO 2 (4,0 pontos)

Em matemática recreacional há um conceito chamado de número feliz <sup>1</sup>.

Para saber se um número é feliz é necessário aplicar o **processo feliz** definido como se segue: Substitua o número pela soma dos quadrados dos seus dígitos e repita o processo até o número ser igual a 1, ou até ele entrar num *loop* infinito, que não incluirá 1. Se o processo chegar no número 1, o número original é um **número feliz**. Se o processo não chegar em 1, ou seja, se o processo entrar em *loop* infinito, o número original é um **número triste**.

Uma propriedade interessante do processo feliz é que, quando ele é aplicado em números tristes, sempre chega-se a um dos números abaixo que começam a se repetir:

4    16    20    37    42    58    89    145

A seguir é exibida a aplicação do processo feliz que descobre que 19 é um número feliz:

$$\begin{aligned}1^2 + 9^2 &= 82 \\8^2 + 2^2 &= 68 \\6^2 + 8^2 &= 100 \\1^2 + 0^2 + 0^2 &= 1\end{aligned}$$

E abaixo a aplicação do processo feliz que descobre que 18 é um número triste:

$$\begin{aligned}1^2 + 8^2 &= 65 \\6^2 + 5^2 &= 61 \\6^2 + 1^2 &= 37 \quad \rightarrow \text{Nesse ponto já é possível concluir que é um número triste} \\3^2 + 7^2 &= 58 \\5^2 + 8^2 &= 89 \\8^2 + 9^2 &= 145 \\1^2 + 4^2 + 5^2 &= 42 \\4^2 + 2^2 &= 20 \\2^2 + 0^2 &= 4 \\4^2 &= 16 \\1^2 + 6^2 &= 37 \quad \rightarrow \text{Repetido, então é um número triste}\end{aligned}$$

Números que são primos e felizes são chamados de **primos felizes**.

Alguns exemplos de primos felizes são:

7 13 19 23 31 79 97 103 109 139 167 193 239 263 293 313 331 367 379 383

---

<sup>1</sup>Veja o episódio “42” de *Doctor Who* onde uma sequência de primos felizes é usada como senha para abrir a porta de uma nave espacial que está prestes a colidir com uma estrela ([http://en.wikipedia.org/wiki/Happy\\_number](http://en.wikipedia.org/wiki/Happy_number)).

(a) (2,0 pontos)

Escreva uma função em python com protótipo

```
def feliz (num):  
    '''(int) -> bool'''
```

que recebe um número inteiro  $num > 0$  e retorna `True` se `num` for um número feliz ou `False` se `num` for um número triste.

```
#-----  
# Solução 1  
def feliz(num):  
    lista = [1, 4, 16, 37, 58, 89, 145, 42, 20]  
    while num not in lista:  
        # calcule o próximo número do processo  
        soma = 0  
        while num != 0:  
            dig = num % 10  
            num //= 10  
            soma += dig*dig  
  
        # soma é o próximo número do processo  
        num = soma  
    return num == 1 # retorna True or False
```

```
#-----  
# Solução 2  
def feliz(num):  
    while num != 1 and num != 4:  
        # calcule o próximo número do processo  
        soma = 0  
        while num != 0:  
            dig = num % 10  
            num //= 10  
            soma += dig*dig  
  
        # soma é o próximo número do processo  
        num = soma  
    return num == 1 # retorna True or False
```

(b) (2,0 pontos) Escreva um programa que lê do teclado um número inteiro  $n > 0$  e uma sequência de  $n$  números positivos como entrada e imprime :-) se **todos** os  $n$  números da sequência forem **primos felizes** :- ( em caso contrário.

Use a função do item (a), mesmo que você não a tenha feito. Use também a função `primo` definida pelo protótipo abaixo (não precisa fazer a função. Considere que ela já existe):

```
def primo (num):  
    '''(int) -> bool'''
```

A função `primo` recebe um número inteiro `num > 0` e devolve `True` se o inteiro `num` é primo e `False` caso contrário.

**Obs.:** Seu programa deve funcionar para **qualquer** sequência, **não** apenas para os exemplos mostrados abaixo.

### Exemplos:

Para  $n=5$  e a sequência 97 7 7 383 139  
o programa deve imprimir :-)

Para  $n=2$  e a sequência 1 2  
o programa deve imprimir :- (

Para  $n=10$  e a sequência 19 23 109 367 379 4 7 13 239 313  
o programa deve imprimir :- (

Para  $n=1$  e a sequência 19  
o programa deve imprimir :-)

#-----

# Solução

```
def main():  
    # leia o tamanho da sequência  
    n = int(input("Digite o tamanho da sequência: "))  
  
    # verifique se cada número da sequência é um primo feliz  
    primos_felizes = True  
    for i in range(n):  
        # leia o proximo número da sequencia  
        num = int(input("Digite um número: "))  
  
        # verifique se num não é feliz ou não é primo  
        if not feliz(num) or not primo(num):  
            # equivalente a: if feliz(num) == False or primo(num) == False:  
            primos_felizes = False  
  
    if primos_felizes:  
        # equivalente a: if primos_felizes == True  
        print(":-)")  
    else:  
        print(":-(")
```

### QUESTÃO 3 (vale 4,0 pontos)

Esta questão consiste na implementação de três funções. Todas as funções são simplificações muito grandes de alguns trechos de código que você escreveu para o seu EP4.

Por um monte de areia entenderemos uma coleção de grãos de areia distribuídos entre as casas de um tabuleiro retangular. Um monte de areia é representado por uma matriz em que cada posição  $[i][j]$  contém o número de grãos na casa. A matriz a seguir representa um monte de areia.

```
      0   1   2   3
+---+---+---+---+
0 | 1 | 1 | 1 | 0 |
+---+---+---+---+
1 | 1 | 12 | 2 | 1 |
+---+---+---+---+
2 | 1 | 1 | 2 | 4 |
+---+---+---+---+
```

As **vizinhas (do tipo rei)** de uma casa são todas as casas que têm em comum com ela pelo menos um bico. Na matriz a seguir o número em cada casa indica o número de vizinhos da casa.

```
      0   1   2   3
+---+---+---+---+
0 | 3 | 5 | 5 | 3 |
+---+---+---+---+
1 | 5 | 8 | 8 | 5 |
+---+---+---+---+
2 | 3 | 5 | 5 | 3 |
+---+---+---+---+
```

Se o número de grãos em uma casa é maior ou igual a quantidade de vizinhos que ela possui, então dizemos que a casa está **instável**. Um monte de areia evolui ao longo do tempo da seguinte maneira. A cada instante, cada casa instável deve espalhar seus grãos entre as suas casas vizinhas, **apenas 1 grão** para cada casa vizinha. Todas as casas instáveis devem espalhar seus grãos **simultaneamente**.

No monte de areia mostrado abaixo à esquerda a casa  $[1][1]$  está instável pois ela possui 12 grãos e tem 8 vizinhos. De maneira semelhante, a casa  $[2][3]$  está instável pois possui 4 grãos e tem 3 vizinhos. Após o espalhamento simultâneo dos grãos das casas instáveis obtemos o monte de areia mostrado abaixo à direita. Nesse novo monte de areia todas as casas estão estáveis.

```
      0   1   2   3
+---+---+---+---+
0 | 1 | 1 | 1 | 0 |
+---+---+---+---+
1 | 1 | 12 | 2 | 1 |
+---+---+---+---+
2 | 1 | 1 | 2 | 4 |
+---+---+---+---+

      0   1   2   3
+---+---+---+---+
0 | 2 | 2 | 2 | 0 |
+---+---+---+---+
1 | 2 | 4 | 4 | 2 |
+---+---+---+---+
2 | 2 | 2 | 4 | 1 |
+---+---+---+---+
```

(a) (vale 1,0 ponto) Escreva uma função

```
def vizinhos(i,j,nlin,ncol):  
    ''' (int,int,int,int) -> lista '''
```

que recebe como parâmetro a posição [i][j] de uma matriz de dimensão  $nlin \times ncol$  e cria e retorna uma lista com as posições vizinhas de [i][j].

**Exemplos** de execuções da função no Python Shell:

```
>>> vizinhos(0,0,3,4)  
[[0, 1], [1, 0], [1, 1]]  
>>> vizinhos(2,2,4,4)  
[[1, 1], [1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2], [3, 3]]  
>>> vizinhos(0,2,4,4)  
[[0, 1], [0, 3], [1, 1], [1, 2], [1, 3]]  
>>> vizinhos(3,2,4,4)  
[[2, 1], [2, 2], [2, 3], [3, 1], [3, 3]]  
>>> vizinhos(0,3,4,4)  
[[0, 2], [1, 2], [1, 3]]  
>>> vizinhos(0,0,1,1)  
[]  
>>>
```

```
#-----  
# Solução
```

```
def vizinhos(i, j, nlin, ncol):  
    lista = []  
    for x in range(i-1,i+2,1):  
        for y in range(j-1,j+2,1):  
            if 0 <= x < nlin and 0 <= y < ncol and (x != i or y != j):  
                lista.append([x,y])  
  
    return lista
```

Para escrever a função pedida no **item (b)** você deve usar a função a seguir sem escrevê-la. Suponha que lhe é dada uma função de protótipo

```
def copia(m):  
    ''' (matriz) -> matriz '''
```

que recebe como parâmetro uma matriz `m` e cria e retorna uma cópia da matriz `m`.

No **item (b)** você deve utilizar ainda a função `vizinhos` do **item (a)** sem reescrevê-la. Você pode usar a função do item (a) mesmo que não a tenha feito.

**(b) (vale 2,0 pontos)** Escreva uma função

```
def espalhe(tab):  
    ''' (matriz) -> matriz '''
```

que recebe uma matriz `tab` que representa um monte de areia e cria e retorna uma matriz `novo_tab` que representa o monte de areia resultante após todas as casas instáveis de `tab` espalharem os seus grãos entre os seus vizinhos.

Por exemplo, se `tab` é a matriz

```
      0  1  2  3  
    +---+---+---+---+  
0 | 5 | 5 | 0 | 0 |  
  +---+---+---+---+  
1 | 1 | 5 | 0 | 1 |  
  +---+---+---+---+
```

então a função deve retornar a matriz

```
      0  1  2  3  
    +---+---+---+---+  
0 | 4 | 2 | 2 | 0 |  
  +---+---+---+---+  
1 | 4 | 2 | 2 | 1 |  
  +---+---+---+---+
```

Atenção, a moldura e os índices não fazem parte das matrizes.

```
#-----  
# Solução  
  
def espalhe(tab):  
    novo_tab = copia(tab) # cria um clone de tab  
    nlin = len(tab)      # número de linhas de tab  
    ncol = len(tab[0])   # número de colunas de tab  
  
    # percorra cada posição de tab  
    for i in range(nlin):  
        for j in range(ncol):  
            # verifique se graos em [i][j] devem ser espalhados  
            lista = vizinhos(i,j,nlin,ncol)  
            n_viz = len(lista)  
            if tab[i][j] >= n_viz:  
                # espalhe os grão da posição [i][j]  
                novo_tab[i][j] -= n_viz  
                for k in range(n_viz):  
                    lin = lista[k][0]  
                    col = lista[k][1]  
                    novo_tab[lin][col] += 1  
  
    return novo_tab
```



Para escrever a função `main` pedida no **item (c)** você **deve** usar as funções a seguir **sem escrevê-las**. Suponha que lhe é dada uma função de protótipo

```
def leia_monte():
    ''' (None) -> matriz '''
```

que lê e retorna uma matriz que representa um monte de areia.

Suponha ainda que lhe é dada uma função de protótipo

```
def imprima(tab):
    ''' (matriz) -> None '''
```

que imprime a matriz do monte de areia representado por `tab`.

No **item (c)** você **deve** utilizar ainda a função `espalha` do **item (b)** **sem reescrevê-la**. Você pode usar a função do item (b) mesmo que não a tenha feito.

**(c) (vale 1,0 ponto)** Escreva uma função `main` que leia:

- a representação `tab` de um monte de areia usando a função `leia_monte` e
- do teclado um inteiro positivo `max`.

A função deve imprimir a situação do monte de areia nos instantes  $0, 1, 2, \dots, \text{max}-1$ . Por exemplo, se o monte de areia é como mostrado a seguir no **Instante 0** e `max = 8` a função deve imprimir:

Instante 0:

```
  0  1  2  3
+---+---+---+---+
0 | 16 | 0 | 0 | 0 |
+---+---+---+---+
1 |  0 | 0 | 0 | 1 |
+---+---+---+---+
```

Instante 4:

```
  0  1  2  3
+---+---+---+---+
0 |  5 | 5 | 0 | 0 |
+---+---+---+---+
1 |  1 | 5 | 0 | 1 |
+---+---+---+---+
```

Instante 1:

```
  0  1  2  3
+---+---+---+---+
0 | 13 | 1 | 0 | 0 |
+---+---+---+---+
1 |  1 | 1 | 0 | 1 |
+---+---+---+---+
```

Instante 5:

```
  0  1  2  3
+---+---+---+---+
0 |  4 | 2 | 2 | 0 |
+---+---+---+---+
1 |  4 | 2 | 2 | 1 |
+---+---+---+---+
```

Instante 2:

```
  0  1  2  3
+---+---+---+---+
0 | 10 | 2 | 0 | 0 |
+---+---+---+---+
1 |  2 | 2 | 0 | 1 |
+---+---+---+---+
```

Instante 6:

```
  0  1  2  3
+---+---+---+---+
0 |  2 | 4 | 2 | 0 |
+---+---+---+---+
1 |  2 | 4 | 2 | 1 |
+---+---+---+---+
```

Instante 3:

```
  0  1  2  3
+---+---+---+---+
0 |  7 | 3 | 0 | 0 |
+---+---+---+---+
1 |  3 | 3 | 0 | 1 |
+---+---+---+---+
```

Instante 7:

```
  0  1  2  3
+---+---+---+---+
0 |  2 | 4 | 2 | 0 |
+---+---+---+---+
1 |  2 | 4 | 2 | 1 |
+---+---+---+---+
```

```
#-----  
# Solução  
  
def main():  
    tab = leia_config()  
    max = int(input("Numero de iteracoes: "))  
  
    for i in range(max):  
        print("Instante:", i)  
        imprima(tab)  
        tab = espalhe(tab)
```