Completando algoritmos

1. X é primo ? Muitas vezes, é necessário saber dentro de um programa se um determinado inteiro é ou não é primo (Definição: um número inteiro positivo maior do que 1 é primo se e somente se seus únicos 2 divisores são a unidade e ele mesmo).

Há um imenso capítulo na matemática para descobrir a primalidade de um número sem ser necessário procurar seus divisores, já que este procedimento é muito demorado. Como não existe almoço grátis, estes métodos (muitos baseados no Pequeno Teorema de Fermat), não são exatos, podendo apresentar falsos positivos (dizer que um x é primo, quando ele não o é), mas não falsos negativos (quando ele diz que é composto, x é composto !). A solução é chamar muitas vezes o mesmo método para o mesmo número (variando certas condições). A cada resposta positiva, a chance do erro cometido é dividida por 2. Daí poder-se ir tão longe quanto se queira.

Voltando à primalidade, se os números forem pequenos, a busca pode ser feita pelos métodos baseados no crivo de Eratóstenes, que seguem uma otimização crescente:

1. Variar um divisor entre 2 e $n-1$. Examinar o resto em divisões sucessivas. Se alguma divisão deixar resto zero, x é composto e sair. Ao final, concluir que x é primo.

2. Lembrando que o único par primo é o 2, e excluindo-o da busca, concluir que se n é par, n é composto.

3. Outra melhoria é buscar a divisibilidade por 3, e neste caso, concluir que n é composto.

4. Uma melhoria (a principal) é que como não se quer todos os divisores, (apenas um resolve o problema !), a busca pode se encerrar em $\sqrt{n}$ sem ser necessário chegar até n. Isto diminui sobremaneira a carga computacional do algoritmo.

5. Finalmente, os primos obedecem à regra $p = (6 \times x) \pm 1$, pelo que o passo na busca pode ser de 6 em 6. (Se não acreditar nesta regra, examine todos os primos entre 5 e 100...)

Assim, por exemplo, 17 é primo, pois os únicos números inteiros que o dividem sem deixar resto são o 1 e o 17. Já 12 não é primo, pois seus divisores são o 1, 2, 3, 4, 6 e 12. A seguir um algoritmo (incompleto) escrito em Python. Você deve completá-lo, testá-lo, implementá-lo e finalmente achar as respostas pedidas.

```
def primo(n):
  if n==1:
    return False
  if n<4:
    return True
  if x%2==0:
--------------------------------
  if x<9:
    return True
  if x%3==0:
--------------------------------
  r=math.floor(math.sqrt(n))
--------------------------------
  while f<=r:
    if (n%f)==0 or (n%(f+2))==0:
      return False
    f=f+6
  return True
```

A seguir, algumas execuções corretas desta função:

```
primo(123456789012419) é True
primo(1) é False
primo(2) é True
```

2. A soma dos divisores de n Define-se a função $sigma(n)$ como sendo a soma dos divisores próprios de n (ou seja excluindo-se dessa soma o próprio n). Assim, como os divisores de 12 são 1,2,3,4,6 e 12, $sigma(12) = 1+2+3+4+6 = 16$. O algoritmo incompleto em Python é

```
def sigma(n):
  divs=1
-----------------
  while i<n:
    if n%i==0:
-----------------
-----------------
```

Execuções corretas:

```
sigma(12)=16
sigma(123456) = 203696
sigma(1000000) = 1480437
```

3. Criação de uma ABP Uma Árvore Binária de Pesquisa (ABP) é uma interessante estrutura de dados que permite efetuar busca binária em um conjunto de dados SEM que eles precisem estar (ou ser mantidos) em ordem crescente. Nesta implementação não se usarão ponteiros e sim cursores. Conceitualmente é a mesma coisa, e ao custo de uma pequena ineficiência, ganha-se simplicidade, visibilidade, facilidade de depuração e sobretudo tempo de programação. Em uma ABP, a raiz é uma das chaves de busca e ela divide o universo em 2 pedaços: itens menores que a chave estarão vinculados ao filho esquerdo e itens maiores ao filho direito. Acompanhe no exemplo:

```
Seja incluir os números:
     14 4 15 7 74 49 33 64 35 22
```

Eis como ficará esta ABP:

```
0=  1  2 14          <14>
1= -1  3  4     +-----+--+
2= -1  4 15   <4>       <15>
3= -1 -1  7     +--+      +--------------------+
4=  5 -1 74       <7>                         <74>
5=  6  7 49                               +------+
6=  9  8 33                             <49>
7= -1 -1 64                        +-----+--+
8= -1 -1 35                      <33>      <64>
9= -1 -1 22                   +--+--+
                            <22>  <35>
```

Depois que a árvore está criada, para verificar a correção da mesma, vis-a-vis o gabarito, ela deve ser percorrida em largura. No exemplo, daria como resposta: 14, 4, 15, 7, 74, 49, 33, 64, 22 e 35. O algoritmo python é

```
import numpy as np
def crarvore(x):
  arv=np.zeros((len(x),3),int)
  arv[0]=[-1,-1,x[0]]
  i=1
  ult=0
  while i<len(x):
    ult=ult+1
    arv[ult]=[-1,-1,x[i]]
    onde=0
    while onde!=-1:
------------------------------
      if x[i]>arv[onde,2]:
        esqdir=1
------------------------------
      else:
        esqdir=0
------------------------------
    arv[pai,esqdir]=ult
    i=i+1
  return arv
```

Para obter a visitação em largura basta percorrer o desenho da árvore em fatias horizontais. Mais alguns exemplos:

```
15 68 79 14 74 67 43 87 39 96 32 92 27 76,
     deu como décimo = 76

76 8 3 80 58 24 32 13 19 36 51 47 38,
     deu como décimo = 36

50 66 76 9 81 8 93 67 39 35 31,
     deu como décimo = 31
```

4. Egípcios Os egipcios antigos, há mais de 5.000 anos, já conheciam o conceito de π – não com esse nome, claro, mas como a constante que iguala o comprimento de uma circunferência ao seu raio. Como eles não dominavam a aritmética real que temos hoje, π era representado por uma fração ordinária, nomeadamente 22/7. Seu problema aqui é receber um número real qualquer com bastantes casas decimais e localizar qual a fração de números inteiros que chega mais perto do valor do número real original. Ambos - numerador e denominador - devem ser menores do que 100 (ou seja, com 2 dígitos). Eis o código Python

```
def egipcio(nr,lim): # lim deve ser 100
  minimo=+999999
  i=1
  while i<lim:
-----------------------------------------------
    while j<lim:
      ca=i/j
      if (abs(nr-ca))<minimo:
-----------------------------------------------
        mini=i
        minj=j
-----------------------------------------------
    i=i+1
  return [mini,minj]
```

Eis alguns exemplos:

```
9.333090, teve como fração mnemônica 28 / 3
1.23456, deu 79 / 64
6.10987, deu 55 / 9
```

☞ Para você fazer

A seguir, duas instâncias de cada exercício. Você deve executar o algoritmo sobre estes valores e achar a resposta pedida:

1.X é primo ? Responda 1 se X for primo e responda 0 senão.

```
1903471
1915855
```

2. Soma divisores Ache a soma dos divisores de

```
1324067
1584294
```

3. ABP Responda qual o décimo elemento na visitação em largura da árvore resultado da criação com estes números na ordem em que eles aparecem

```
82 20 64 43 28 78 59 88 4 27 76 56 37
36 83 79 22 15 73 72 56 69 98 58 87
```

4. Egípcios Ache o numerador e denominador inteiros, menores que 100 que mais se aproximam do número dado.

```
0.387755102
0.5869565217
```

1.1	1.2	2.1	2.2

3.1	3.2	4.1	4.2