Hoja de ejercicios 3

Bucles

13 de enero de 2016

Informática
Año 2015/2016
Facultad de CC.
Matemáticas

▷ 1. Número de cifras en una determinada base

Escribe un programa que determine cuántas cifras tiene una cantidad C si la expresamos en una determinada base b.

Solución

Una solución sencilla consiste en ir eliminando la cifra menos significativa de un número sucesivamente hasta llegar a una única cifra. Es decir, si nos dan un número, por ejemplo el 3754, podemos comprobar si tiene una cifra o no, si tiene más de una cifra entonces podemos eliminar la última (el 4) y añadir uno al contador de cifras, de esta forma nos queda el 375 y seguimos contando sus cifras de igual manera. Esto se expresa mejor en un programa:

```
def numCifras(m,base):
    cont=1
    while m >= base:
        m = m / base
        cont = cont +1
    return cont
```

El programa anterior es *bastante* correcto, pero es fácilmente mejorable. ¿Se te ocurre cómo? (Piensa un poco antes de seguir...) En primer lugar, el código que acabamos de escribir no funciona correctamente con números negativos. Eso se soluciona fácilmente. Además, siempre que aparezca una constante numérica en el código, como en este caso es el 10, deberíamos preguntarnos que papel juega. En este caso, el 10 es fruto de una suposición no exigida en el enunciado: estamos suponiendo que los números están expresados en base 10. De hecho, con el mismo esfuerzo podríamos solucionar el problema de contar las cifras de una determinada cantidad expresada en cualquier base. El código siguiente tiene en cuenta estas mejoras:

```
def numCifras(m,base):
    cont=1
    m = abs(m)
    while m >= base:
        m = m / base;
        cont = cont +1;
    return cont
```

Observa la instrucción `int m = abs(n)` que es la que se encarga de hacer que nuestro programa pueda funcionar con números negativos.

▷ 2. Reverso de un número

Escribe un programa que "dé la vuelta" a un número. Por "dar la vuelta" entendemos invertir las cifras, por ejemplo: el número 17 deberá convertirse en el 71, el 120 en el 21,...

Solución

Una solución sencilla consiste en ir extrayendo la cifra menos significativa del número, e ir construyendo a la vez el reverso, actualizando su valor por cada cifra que obtenemos. Esto se expresa mejor en un programa:

```
def reverse_num(number):
    reverse = 0
    while number != 0:
        reverse = 10*reverse + (number % 10)
        number = number / 10
    return reverse
```

▷ 3. Suma marciana

Se ha encontrado en Marte la siguiente operación de sumar, resuelta en una roca:

$$\begin{array}{ccc} \clubsuit & \diamondsuit & \spadesuit \\ & \clubsuit & \heartsuit \\ \hline \diamondsuit & \spadesuit & \clubsuit \end{array}$$

Se desea descifrar el significado (o sea, el valor) de esos símbolos, suponiendo que se ha empleado el sistema de numeración decimal.

Solución

Una posibilidad consiste en producir cada una de las combinaciones posibles,

♣	♢	♠	♡
0	0	0	0
0	0	0	1
...	...	...	...
0	0	0	9
0	0	1	0
...	...	...	...
0	9	9	9
1	0	0	0
...	...	...	...

y examinar cuáles de ellas verifican esa cuenta.

Para concretar un poco más el problema, tendremos en cuenta lo siguiente:

- Por estar en el sistema de numeración decimal, los posibles valores de cada uno de los símbolos que intervienen en la cuenta son los valores del cero al nueve.
- Como se ha empleado el sistema de numeración decimal, la grafía ♣♢♠ representa la cantidad $100\clubsuit + 10\diamondsuit + \spadesuit$.

Entonces, el algoritmo descrito se traduce a Python fácilmente así:

```
def martian_sum():
    club = 0
    while club <10:
```

```
diamond = 0
while diamond<10:
    spade = 0
    while spade<10:
        heart = 0
        while heart<10:
            sum1 = 100*club + 10*diamond + spade
            sum2 = 10*club + heart
            sum = 100 * diamond + 10 * spade + club
            if sum1+sum2 == sum:
                print "  ", heart,diamond,spade
                print "+   ",club,heart
                print "---------"
                print "  ",diamond,spade,club
            else:
                print sum1,sum2,sum
            heart +=1
        spade +=1
    diamond += 1
club += 1
```

Ahora, pueden hacerse dos observaciones que nos permiten limitar un poco los tanteos:

- Los símbolos ♣ y ♢ no pueden ser nulos, ya que están al principio de los números.
- Los cuatro símbolos empleados por los habitantes de Marte pueden suponerse distintos.

▷ 4. Calculando sobre listas

Especifica y diseñar algoritmos para:

- Llenar una lista con datos aleatorios.
- Mostrar los valores de una lista por pantalla.
- Calcular la media y la desviación típica de los elementos de la lista.
- Calcular la cantidad de números pares.
- Calcular la cantidad de cuadrados perfectos.
- Calcular la cantidad de los números cuyo logaritmo en base 2 sea menor que otro número dado `log` > 0.
- Determinar si la lista está ordenada de menor a mayor.
- Contar el número de *picos* que contiene. Un número es un pico si es estrictamente mayor que los dos que tiene a su lado.
- Calcular la cantidad de números que sean potencia de dos.
- Decidir si la lista es *palíndroma*, es decir, puede leerse igual de izquierda a derecha que de derecha a izquierda.

- Calcular la cantidad de números primos que contiene.
- Calcular el mínimo y el máximo de una lista.
- Si el nombre de la lista es `t`, un algoritmo que calcule en otra tabla `s` las sumas parciales de los elementos de la primera:

$$s[1] = t[1],\ s[2] = t[1] + t[2],\ s[3] = t[1] + t[2] + t[3],\ \cdots \quad s[i] = \sum_{k=1}^{i} t[k]$$

Solución

- Llenar una lista con datos aleatorios.

```
def random_list(size) :
    """
    Create a list of "size" random integers.

    Parameters:
    ------------
    size : int
    size > 0

    Returns
    -------------
    [int]
    list of random numbers

    Example
    --------------
    >>> random_list(9)
    [42, 90, 73, 66, 64, 81, 19, 21, 27]
   """
    i,l = 0,[]
    while i < size :
        l.append(random.randint(0,100))
        i = i + 1
    return l
```

- Mostrar los valores de una lista por pantalla.

```
def show (list) :
    i = 0
    n = len (list)
    print "[",
    while i < n - 1 :
        print str(list[i]) + ",",
        i += 1
    print str(list[n - 1]) + "]"
```

- Calcular la media y la desviación típica de los elementos de la lista.

```
def average(l):
    """
    Computes average of a given list of integers

    Parameters:
    ------------
    l : [int]
      non-empty

    Returns
    -------------
    float
        average of input elements

    Example
    --------------
    >>> average([5,10])
    7.5
   """
    s,i=0.0,0
    while (i<len(l)):
        s = s + l[i]
        i = i + 1
    return s / len(l)

def variance(l):
    """
    Computes variance of a given list of values.  Recall that this is
    the average of the squared differences from the Mean.

    Parameters:
    ------------
    l : [int]
      non-empty

    Returns
    -------------
    float
        variance  of input elements

    Example
    --------------
    >>> variance([1,2,3,4,5,5])
    2.2222222222222223
   """
    avg=average(l)
    r,i=[],0
    while i < len(l):
        r.append((avg-l[i])**2)
        i = i + 1    # i += 1
    return average(r)
```

- Calcular la cantidad de números pares.

```
def countForEven(t) :
    """
    Computes number of even numbers

    Parameters:
    ------------
    t : [int]

    Returns
    -------------
    int
        number of even numbers

    Example
    --------------
    >>> countForEven([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])
    7
   """
    i,cont = 0, 0
    while i < len(t):
        if (t[i] % 2 == 0):
            cont = cont + 1
        i = i + 1
    return cont
```

- Para contar los número que verifican la propiedad de ser cuadrados perfectos tenemos que recorrer todo el vector, para ello lo más sencillo es utilizar un bucle `for`.

```
import math
def countPerfectSquare(t):
    count=0
    for n in t:
        sr = round(math.sqrt(n))
        if n == sr*sr:
            count +=1
    return count
```

- Calcular la cantidad de los números cuyo logaritmo en base 2 sea menor que otro número dado $\texttt{log} > 0$.

```
def logs (t, log_value) :
    i = 0
    n = len (t)
    cont = 0
    while i <= n - 1 :
        if (log(t[i]) / log(2)) < log_value :
            cont += 1
        i += 1
    return cont
```

- Determinar si la lista está ordenada de menor a mayor. Para decidir si la tabla se encuentra ordenada utilizamos un bucle `while` puesto que si nos damos cuenta de que un par de posiciones no están ordenadas entonces detenemos el recorrido.

```
def sorted(t):
    i=0
    n=len(t)
    while i<n-1 and t[i]<=t[i+1]:
        i +=1
    return i>=n-1
```

- Contar el número de *picos* que contiene.

```
def peaks(lst):
    count = 0
    i = 1 #the first element cannot be a peak
    while i<len(lst)-1: # the last element cannot be a peak
        if lst[i]>lst[i-1] and lst[i]>lst[i+1]:
            count += 1
        i += 1
    return count
```

- Calcular la cantidad de números que sean potencia de dos.

```
def countForPower2(l):
    """
    Counts the number of 2-powers in
    between those elements (ints) in the list

    NOTE: For didactical purposes, see the nested loop
    (usually more abstraction should be given on a separate routine)
    Parameters:
    ------------
    t : [int]
        ints are strictly greater than zero!!!

    Returns
    -------------
    int
        number of 2-powers

    Example
    --------------
    >>> countForPower2([])
    0
    >>> countForPower2([12,2,2,4])
    3
    >>> countForPower2([4,4,4,4,4])
    5
    >>> countForPower2([3,3,17,25,3])
    0
    """
    i,cont=0,0
```

```
    while (i < len (l)):
        n = l[i]
        while ((n%2)==0):
            n = n / 2
        if (n==1):
            cont = cont + 1
        i = i + 1
    return cont
```

- Decidir si la lista es *palíndroma*, es decir, puede leerse igual de izquierda a derecha que de derecha a izquierda.

```
def palyndrome(word):
    """
    Check if a word is the same when
    readen from left to right and
    from right to left

    Parameters:
    ------------
    t : [char] i.e. , str
         true (i.e., empty, even or odd size!)
    Returns
    -------------
    bool
        true when description above fits

    Example
    --------------
    >>> palyndrome('')
    True
    >>> palyndrome('aba')
    True
    >>> palyndrome('abd')
    False
    >>> palyndrome('dabalearrozalazorraelabad')
    True
    """
    i,half=0,(len(word)-1)/2
    n = len(word)
    while (i <= half) and (word[i]==word[n-i-1]):
        i = i + 1
    return i>half
```

- Calcular la cantidad de números primos que contiene.

```
def countForPrimes(l):
    """
    Very hard. Recall Prime Theorem:
    the last prime number up to itself is at most as its half
    NOTE: For didactical purposes, use a nested loop.
    Usually more abstraction is required.
```

```
    Parameters:
    ------------
    t : [int]
        list of ints (possibly empty) _strictly_ greater than 1

    Returns
    -------------
    int
        number of prime numbers

    Example
    --------------
    >>> countForPrimes([2,3,4,5,6,7,8,9,10,11,12])
    5
    >>> countForPrimes([10,11])
    1
    >>> countForPrimes([11,11])
    2
    >>> countForPrimes([11,11,14,27])
    2
    """
    cont,j = 0,0
    while (j<len(l)):
        i,n = 2,l[j]    # More abstraction here. Pull out this computation
        while ((i<=(n/2)) and (n%i != 0)):
            i=i+1
        if (i > (n / 2)):
            cont = cont + 1
        #Loop Step
        j = j + 1
    return cont
```

- Calcular el mínimo y el máximo de una lista.

```
def minMax(l):
    """
    Computes maximum and minimum of
    a given list of integers
    ------------
    l : [int]
        size of l greater than 0, non-empty list!!

    Returns
    -------------
    (int,int)
        a pair recording minimum and maximum

    Example
    --------------
    >>> minMax([4,5])
    (4, 5)
    >>> minMax([4,5,25,45,789])
    (4, 789)
```

```
    """
    i, minimum, maximum=1,l[0],l[0]
    while (i < len(l)):
        if (l[i] > maximum):
            maximum = l[i]
        if (l[i] < minimum):
            minimum = l[i]
        i = i + 1
    return (minimum, maximum)

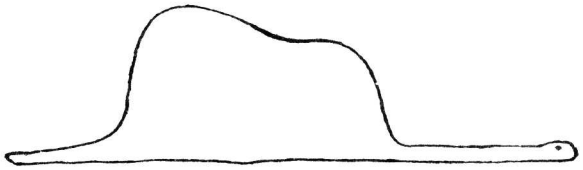

def partialSums(l):
    """
    Computes partials sums. i.e.
    r[i]= sum[0..i]{l[i]}
    ------------
    l : [int]
        true, even empty lists!

    Returns
    -------------
    [int]
        list of partial sums.

    Example
    --------------
    >>> partialSums([])
    []
    >>> partialSums([1,1,12])
    [1, 2, 14]
    >>> partialSums([1,2,12])
    [1, 3, 15]
    """
    i,s=0,0
    r=[None]*len(l)  # This simulates a declared vector in Pascal. Explain
    while (i<len(l)):
        r[i],s = s + l[i], s + l[i]   #!!! Very purist!
        i=i+1
    return r
```

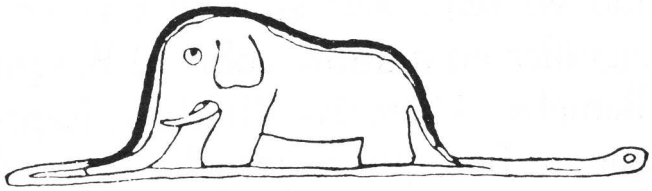
▷ 5. El principito

- Mi dibujo número uno. Era así:



Mostré mi obra maestra a las personas mayores y les pregunté si mi dibujo les daba miedo. Me contestaron: "¿Por qué nos habría de asustar un sombrero?"

Pero mi dibujo no representaba un sombrero, sino una serpiente boa que digería un elefante. Dibujé entonces el interior de la serpiente boa, a fin de que las personas adultas pudiesen comprender, pues los adultos siempre necesitan explicaciones. Mi dibujo número dos era así:



Las personas mayores me aconsejaron abandonar los dibujos de serpientes boas abiertas o cerradas y que pusiera más interés en la geografía, la historia, el cálculo y la gramática. Y así fue como a la temprana edad de seis años, abandoné una magnífica carrera de pintor, desalentado por el fracaso de mis dibujos números uno y dos. Las personas mayores nunca comprenden por sí solas las cosas, y resulta muy fastidioso para los niños tener que darles continuamente explicaciones.

El principito, Antoine de Saint-Exupéry

Al principito le gustan mucho los números y prefiere las fracciones así:

$$\frac{5687171}{18686419}$$

ya que puede ver y disfrutar de muchas cifras. Sin embargo los mayores prefieren ver cosas más simples y explicadas:

$$\frac{5687171}{18686419} = \frac{7 \cdot 812453}{23 \cdot 812453} = \frac{7}{23}$$

Pista: Se trata de simplificar al máximo una fracción $\frac{n}{d}$. Para ello, tenemos que calcular el máximo común divisor del numerador y del denominador, $\text{mcd}(n, d)$, que es el mayor número por el que podemos dividir tanto n como d.

Solución

Para no tener que dar explicaciones a los mayores el principito ha decidido hacer un programa que simplifique las fracciones y así, si alguien viene y se asusta de ver largas ristras de números, el programa se encargará de mostrar la fracción más reducida posible.

Para ello, como aprendió en sus primeras clases de matemáticas, basta con calcular el máximo común divisor entre el numerador y el denominador y dividir tanto el numerador como el denominador por dicho número. Es decir, si $m = \mathrm{mcd}(n, d)$ y $n' = \frac{n}{m}$ y $d' = \frac{d}{m}$, entonces $\frac{n}{d} = \frac{n'}{d'}$ y la fracción $\frac{n'}{d'}$ es irreducible.

Notas bibliográficas En el siglo III a. C., Euclides escribió los *Elementos* [Euc91, Euc94, Euc96], dividida en trece volúmenes, que ha sido la obra matemática por excelencia durante más de dos mil años. En su libro VII [Euc94], aparece el conocido algoritmo de Euclides para encontrar el máximo común divisor de dos números. En [Cha99] también se relata el texto original de Euclides explicando el algoritmo. Puede encontrarse la implementación de este algoritmo, y de otros similares, en infinidad de libros básicos de programación, como [BB00].

▷ 6. Descomposición en factores primos

Seguro que has realizado alguna vez la tediosa tarea de descomponer un número en factores primos:

12	2	85	5	56	2	110	2
6	2	17	17	28	2	55	5
3	3	1		14	2	11	11
1				7	7	1	
				1			

¿Podríamos hacer un programa que fuése capaz de hacer estas descomposiciones? Seguro que sí. Escribe un programa que reciba como entrada un número entero positivo y calcule la descomposición del mismo en factores primos.

Solución

Una solución básica es la siguiente:

```
def factors(n):
    """
    This function prints the list of factors of n
    @type n: int
    @param n: n>1
    @rtype: List
    """
    fct = 2
    factors = []
    while n>1:
        if n % fct == 0:
            factors.append(fct)
            n = n/fct
        else:
            fct += 1
    return factors

def factors_exp(n):
    """
    This function returns the list of factors of n,
    together with their exponents
    @type n: int
    @param n: n>1
    @rtype: List
```

```
    """
    fct = 2
    factors = []
    while n>1:
        if n % fct == 0:
            exp = 1
            n = n/fct
            while n % fct ==0:
                exp += 1
                n = n/fct
            factors.append([fct,exp])
        else:
            fct += 1
    return factors

def main():
    cont = "s"
    while cont=="s":
        n = int(raw_input("Dame un únmero: "))
        if n>1:
            print "La lista de factores es: ", factors(n)
            print "La lista de factores es: ", factors_exp(n)
        else:
            print "El únmero debe ser mayor que 1"
        cont = raw_input("Otro [s/N]")
```

Si se desea una presentación algo más bonita podemos mostrar los factores comunes en forma de potencia y añadir el símbolo del producto:

```
c='s'
while c!='n':
  n=int(raw_input( 'Numero a descomponer? '))
  while n<=1:
    print "El numero ha de ser mayor que 1"
    n=int(raw_input( 'Numero a descomponer? '))
  print 'Factores primos:'
    i=2
    while n>1:
      mult = 0
      while (n%i==0):
          mult = mult + 1
          n=n/i
      if mult>0:
        if n>1:
          print str(i)+'^'+str(mult)+'*'
        else:
          print str(i)+'^'+str(mult)
      i= i + 1

  c=raw_input('Otro numero (s/n)?')
```

▷ 7. Los cubos de Nicómaco

(*) Considera la siguiente propiedad descubierta por Nicómaco de Gerasa:

Sumando el primer impar, se obtiene el primer cubo;
sumando los dos siguientes impares, se obtiene el segundo cubo;
sumando los tres siguientes, se obtiene el tercer cubo, etc.

Comprobémoslo:

$$\begin{array}{lcll} 1^3 & = & 1 & = & 1 \\ 2^3 & = & 3+5 & = & 8 \\ 3^3 & = & 7+9+11 & = & 27 \\ 4^3 & = & 13+15+17+19 & = & 64 \end{array}$$

Desarrolla un programa que calcule los n primeros cubos utilizando esta propiedad.

Un poco de historia Nicómaco de Gerasa vivió en Palestina entre los siglos I y II de nuestra era. Escribió *Arithmetike eisagoge* (Introducción a la aritmética) que fue el primer tratado en el que la aritmética se consideraba de forma independiente de la geometría. Este libro se utilizó durante más de mil años como texto básico de aritmética, a pesar de que Nicómaco no demostraba sus teoremas, sino que únicamente los ilustraba con ejemplos numéricos.

Solución

Utilizaremos la variable `impar` para que vaya tomando los valores de los números impares. Su valor inicial será 1 ya que así, al incrementarla sucesivamente en 2 unidades, se irán generando los valores impares. Debemos calcular los n primeros cubos:

```
for i in range (1,n+1):
  # Calculo del cubo /i/-esimo a partir de /i/ numeros impares
```

Sabemos que el primer cubo se calcula sumando el primer impar; el cálculo del cubo i-ésimo, cuando $i > 1$, se realiza sumando los i siguientes impares; necesitaremos, por lo tanto, además de ir generando números impares consecutivos (`impar += 2`), una variable que vaya acumulando su suma (`suma += impar`) y un bucle que controle que estas operaciones se realizan el número adecuado de veces.

```
impar = 1;
for i in range(1,n+1):
    print i,"^ 3 = ",
    suma = 0
    for j in range(i):
        suma += impar
        if j>0:
            print "+",
        print impar,
        impar += 2
    print "=",suma
```

▷ 8. Ecuación diofántica

Se llama ecuación *diofántica* a cualquier ecuación algebraica, generalmente de varias variables, planteada sobre el conjunto de los números enteros $\mathbb{Z}$ o los números naturales $\mathbb{N}$.

- Escribe un programa que calcule las maneras diferentes en que un número natural n se puede escribir como suma de otros dos números naturales. Es decir, que calcule las soluciones de la ecuación diofántica $x + y = n$.

- Escribe un programa que calcule las maneras diferentes en que un número natural n se puede escribir como producto de dos números naturales. Es decir, que calcule las soluciones de la ecuación diofántica $xy = n$.

- Escribe un programa que, dado un número natural n, calcule la cantidad de soluciones de la ecuación diofántica: $x^2 - y^2 = n$

Pista: Ten en cuenta que toda solución de la anterior ecuación nos produce una factorización del entero n:

$$n = (x + y)(x - y)$$

Ya sólo nos queda dilucidar cuáles de esas factorizaciones producen una solución de la ecuación.

Un poco de historia La palabra *diofántica* hace referencia al matemático griego del siglo III de nuestra era Diofanto de Alejandría.
(`http://es.wikipedia.org/wiki/Diofanto_de_Alejandr%C3%ADa`)

Solución

-
```
def diof_sum(n):
    """
    This function computes the natural solutions
    to the equation x + y = n
    @type n: int
    @precondition: 0 <= n
    """
    sols = []
    for x in xrange(n+1):
        sols.append((x, n-x))
    return sols


def diof_prod(n):
    """
    This function computes the natural solutions
    to the equation x * y = n
    @type n: int
    @precondition: 0 <= n
    """
    sols = []
    for x in xrange(1,n+1):
        if n%x==0:
            sols.append((x, n//x))
    return sols
```

- Para calcular las soluciones de $x^2 - y^2 = n$, puesto que $x^2 - y^2 = (x-y)(x+y)$, primero buscamos todas las soluciones de $z \cdot t = n$. Ahora para cada posible solucion de $x+y = t$, comprobamos si $x - y = z$ y en ese caso hemos encontrado una solción. Además no hay otras posibles soluciones.

```
def diof_catethus(n):
    """
    This functions computes the natural solutions
    to the equation x-y = n
    @type n: in
    @precondition: n>=0
    """
    sols = []
    # x - y = (x-y)*(x+y)
    # so we first computes all possible solutions
    # to minus * plus = n
    for (minus, plus) in diof_prod(n):
        # We compute all solutions x + y = plus
        for (x, y) in diof_sum(plus):
            # If x - y = minus, we have found a solution
            if x - y == minus:
                sols.append((x, y))
    return sols
```

▷ 9. Aproximaciones al número π

Desde que el ser humano se ha preocupado por conocer el entorno y explicar el porqué de las cosas que lo rodean, ha habido personas que han intentado calcular la relación existente entre la longitud de la circunferencia y el radio (o diámetro) que la define.

Largo ha sido el periplo de los matemáticos en torno a este número. En este ejercicio te proponemos utilizar las siguientes fórmulas matemáticas para construir programas que permitan calcular aproximaciones al número π.

- François Viète (1540–1603) en 1593:

$$\frac{2}{\pi} = \sqrt{\frac{1}{2}} \cdot \sqrt{\frac{1}{2} + \frac{1}{2}\sqrt{\frac{1}{2}}} \cdot \sqrt{\frac{1}{2} + \frac{1}{2}\sqrt{\frac{1}{2} + \frac{1}{2}\sqrt{\frac{1}{2}}}} \cdots$$

- John Wallis (1616–1703) en 1656:

$$\frac{4}{\pi} = \frac{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \cdots}{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \cdots}$$

- Gottfried Wilhelm Leibniz (1646–1716) en 1673:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

- Borwein en 1987:

$$x_0 = \sqrt{2} \qquad y_1 = 2^{\frac{1}{4}} \qquad \pi_0 = 2 + \sqrt{2}$$

$$x_{n+1} = \frac{1}{2}\left(\sqrt{x_n} + \frac{1}{\sqrt{x_n}}\right) \qquad y_{n+1} = \frac{y_n\sqrt{x_n} + \frac{1}{\sqrt{x_n}}}{y_n + 1} \qquad \pi_n = \pi_{n-1}\frac{x_n + 1}{y_n + 1}$$

Tiene una convergencia muy rápida: $\pi_n - \pi < 10^{-2^{n+1}}$.

Notas bibliográficas π es sin duda el más famoso de los números, y por eso la bibliografía sobre él es extensísima. Dos libros llenos de curiosidades sobre π son [AH01] y [Bec82]. En [Cha99] se dedica un capítulo a los diversos métodos usados a lo largo de la historia para calcular π. Tan distinguido número no podía faltar tampoco en Internet:

```
http://www.joyofpi.com/pilinks.html
http://www-groups.dcs.st-and.ac.uk/~history/HistTopics/Pi_through_the_ages.html
```

En las dos primeras direcciones hay cientos de referencias diversas dedicadas a π; en la tercera puedes encontrar muchas, pero que muchas, cifras decimales de π.

Un poco de historia El primero que utilizó el símbolo π fue William Jones (1675–1749) en 1706. A Euler le gustó este símbolo, lo adoptó y difundió su uso. La fórmula de Leibniz es una particularización de la serie que define el arcotangente de un ángulo; James Gregory (1638–1675) la había descrito con anterioridad, pero no hay ninguna información de que la usase para aproximar el número π.

He aquí una tabla con la cronología del número de cifras decimales de π calculadas:

Número de decimales	Año	Autores	Sistema informático
2 037	1949	G.W. Reitwiesner, ...	ENIAC
10 000	1958	F. Genuys	IBM 704
100 265	1961	D. Shanks y J. Wrench	IBM 7090
1 001 250	1974	J. Guilloud y M. Bouyer	CDC 7600
16 777 206	1983	Y. Kanada, S. Yoshino y Y. Tamura	HITAC M-280H
134 214 700	1987	Y. Kanada, Y. Tamura, Y. Kubo, ...	NEC SX-2
6 442 450 000	1995	D. Takahashi y Y. Kanada	HITAC S-3800/480 (2 procesadores)
51 539 600 000	1997	D. Takahashi y Y. Kanada	HITACHI SR2201 (1024 procesadores)
206 158 430 000	1999	D. Takahashi y Y. Kanada	HITACHI SR8000 (128 procesadores)

Solución

- ```
  def PI_viete(n):
      factor = 1/sqrt(2)
      producto = factor
      for i in range(n):
          factor = sqrt((1.0/2 + factor/2))
          producto = producto * factor
      return 2/producto


  def PI_wallis(n):
      tempo = 1.0
      k = 2
      for i in range(n):
          tempo = tempo*(float((k+1)**2)/(k*(k+2)))
          k = k + 2
      return 4/tempo
  ```
  ```

-
```
def PI_leibniz(n):
    k = 1
    sumando = 1
    s = sumando
    for i in range(n):
        sumando = 1.0/(2*k + 1)
        if k%2==1: sumando = -sumando
        s = s + sumando
        k += 1
    return 4*s
```

- El método de Borwein para aproximar el valor de π es extemandamente rápido, tan sólo en la segunda iteración $n = 2$, el valor de π_2 difiere de π en menos de 10^{-8}, es decir, menos de una cienmillonésima. Haremos un programa que calcule los valores de la sucesión π_n hasta el valor de n que deseemos.

  El programa no requiere grandes habilidades de programación, basta con ser cuidadosos a la hora de plasmar las dependencias entre los cálculos de las diversas sucesiones implicadas, x_n, y_n y π_n.

```
from math import sqrt

def next_x(x):
    return 0.5*(sqrt(x)+1/sqrt(x))

def next_y(x,y):
    return (y*sqrt(x) + 1/sqrt(x))/(y+1)

def next_pi(x, y, pi):
    return pi * ((x+1)/(y+1))

def borewein(n):
    x = next_x(sqrt(2)) #x=x_1
    y = sqrt(sqrt(2)) #y=y_1
    pi = 2 + sqrt(2) #pi=π_0
    i = 0
    while i<n:
        # INV: pi = π_i, x=x_{i+1}, y=y_{i+1}
        pi = next_pi(x, y, pi) #pi=π_{i+1}
        y = next_y(x,y) #y=y_{i+2}
        x = next_x(x) #x=x_{i+2}
        i += 1
    return pi
```

▷ **10. Método de Newton.**

Desde hace mucho tiempo los matemáticos se han preocupado de hallar soluciones de ecuaciones. De estas las más sencillas son las polinómicas. Si la solución x a un problema cumple $a \cdot x + b = 0$, con a, b números reales la cosa es muy fácil: $x = a/b$. Todo el mundo sabe que si un problema se reduce a resolver una ecuación del tipo $a \cdot x^2 + b \cdot x + c = 0$, con a, b y c números reales, basta aplicar una sencilla fórmula para obtener los dos posibles valores de x.

Para las de grado 3 y 4 existen fórmulas para obtener las raíces pero la cosa ya se pone un poco más complicada (de hecho, salvo raras excepciones, no se estudian esas fórmulas en la educación secundaria).

La situación es peor para las de grado 5, para algunos polinomios existe una fórmula que nos dá las raíces en función de los coeficientes y para otros se sabe que no existe tal fórmula. Demostrar ésto último es una cuestión algebraica avanzada, la wikipedia nos dá buena información sobre esto: `http://en.wikipedia.org/wiki/Quintic_function`

Afortunadamente existen métodos numéricos para aproximar las soluciones de una ecuación. Si tenemos un polinomio $f(x)$ y un x_0, podemos sustituir el polinomio $f(x)$ por la recta tangente a la curva $y = f(x)$ en el punto $(x_0, f(x_0))$. Esa recta corta al eje x en un punto cuya coordenada x es $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$. Si repetimos el proceso obtenemos una sucesión: $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$. Salvo para unas pocas elecciones de x_0 la sucesión converge a un cero de f.

Aproximar ceros. Debes definir una función newton que dados: una función (f), su derivada ($derf$), un valor inicial(x_0), un epsilon y un número máximo de iteracciones(max_iter), devuelva:

- una aproximación de un cero de la función f obtenida aplicando el método de Newton y el número de iteraciones que hemos necesitado. (consideramos que un x_k es suficientemente bueno si abs($f(x_k)$)<epsilon)
- si tras realizar el número maximo de iteraciones la aproximación no es aceptable devolvemos el valor alcanzado y el número de iteraciones.

Solución

```
EPSILON = 1e-10
def newton(f, derf, x0, max_iter):
    """
    This function iterates the newton method to compute
    a solution of f.
    @type f: float->float
    @type derf: float->float
    @type x0: float
    @type max_iter: int
    @rtype (float, int)
    @precondition: derf is the derivative of f
    """
    i = 0
    x_i = x0
    fx_i = f(x0)
    while i<max_iter and abs(fx_i)>EPSILON:
        x_i = x_i - fx_i/derf(x_i)
        fx_i = f(x_i)
        i += 1
    return x_i,i

import math

def log(x):
```

```
    return math.log(x)-1

def der_log(x):
    return 1.0/x

def main():
    enumber, iter = newton(log, der_log, 3.0, 10)
    print enumber, iter
```

▷ 11. Espectro natural

(*) El *espectro natural* de una circunferencia de radio r es el conjunto de puntos con coordenadas naturales (n, m) tales que $n^2 + m^2 = r^2$. El método obvio para calcular el espectro necesitaría dos bucles anidados en donde se irían explorando las abcisas y ordenadas de los puntos desde 0 a r. Afortunadamente existe un método más eficiente, que además aprovecha la simetría del problema: si (n, m) está en el espectro también está (m, n). Definimos

$$B(x, y) = \{(n, m) \mid n^2 + m^2 = r^2 \wedge x \leq n \leq m \leq y\}$$

Obviamente el espectro es $B(0, r) \cup \{(m, n) \mid (n, m) \in B(0, r)\}$ que a su vez se puede calcular usando las siguientes reglas:

$$B(x, y) = \begin{cases} B(x+1, y), & \text{si } x^2 + y^2 < r^2 \\ \{(x, y)\} \cup B(x+1, y-1), & \text{si } x^2 + y^2 = r^2 \\ B(x, y-1), & \text{si } x^2 + y^2 > r^2 \end{cases}$$

Escribe un programa iterativo que calcule el espectro natural de una circunferencia de radio r utilizando exclusivamente las reglas dadas en el párrafo anterior.

Indica el número de pasos que lleva calcular el espectro en relación al radio r.

Solución

El espectro natural de una circunferencia de radio r, centrada en el origen, consiste en los puntos del plano con coordenadas naturales que pertenecen a dicha circunferencia. La figura 1 muestra los puntos del espectro natural de la circunferencia de radio 10. Debido a la simetría

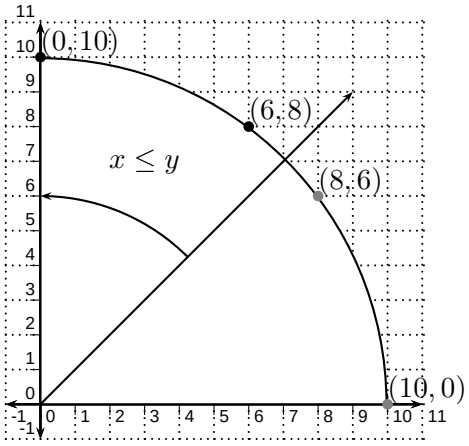


Figura 1: Puntos del espectro natural de una circunferencia de radio 10

del problema, basta con que busquemos los puntos que verifiquen que su coordenada x es menor o igual que su coordenada y. Si definimos el conjunto de puntos

$$B(x,y) = \{(n,m) \mid n^2 + m^2 = r^2 \wedge x \leq n \leq m \leq y\}$$

calcular el espectro de una circunferencia se reduce a encontrar los puntos que pertenecen $B(0,r)$ y para ello se dan las siguientes reglas:

$$B(x,y) = \begin{cases} B(x+1,y), & \text{si } x^2+y^2 < r^2 \\ \{(x,y)\} \cup B(x+1,y-1), & \text{si } x^2+y^2 = r^2 \\ B(x,y-1), & \text{si } x^2+y^2 > r^2 \end{cases}$$

Un programa que solucione el problema debe, por tanto, tratar de seguir estas reglas para encontrar los puntos que pertenecen a $B(0,r)$. Para ello, como indican las reglas, si estamos en el punto (x,y), dependiendo de que dicho punto esté *dentro* de la circunferencia, justo *en* ella, o *fuera* de ella, tendremos que probar con $(x+1,y)$, $(x+1,y-1)$, o con $(x,y-1)$, respectivamente.

```
"""
Programa que calcula el espectro natural de un únmero natural
"""

def espectro(n):
    x = 0
    y = n
    l = []
    while x<=y:
        dist = x*x + y*y
        if dist == n*n:
            l.append((x, y))
            l.append((y, x))
            x = x+1
            y = y-1
        elif dist > n*n:
            y = y-1
        else:
            x = x+1
    return l
```

▷ 12. Criba de Eratóstenes

Un método para encontrar todos los números primos entre 1 y n es la *criba de Eratóstenes*. Considera una lista de números entre 2 y n. El número 2 es el primer primo, pero todos los múltiplos de 2 (4, 6, 8, ...) no lo son, y por tanto pueden ser tachados de la lista. El primer número después de 2 que no está tachado es 3, el siguiente primo. Tachamos entonces de la lista los siguientes múltiplos de 3, por supuesto, a partir de 3×3 ya que los anteriores están ya tachados (9, 12, ...). El siguiente número no tachado es 5, el siguiente primo, y entonces tachamos los siguientes múltiplos de 5 (25, 30, 35, ...). Repetimos este proceso hasta que alcanzamos el primer número en la lista que no está tachado y cuyo cuadrado es mayor que n. Todos los números que quedan en la lista sin tachar son los primos entre 2 y n.

Escribe un programa que utilice este algoritmo de criba para encontrar todos los números primos entre 2 y n.

Solución

```
   ó
Implementacin de la Criba de óEratstenes
'''

def no_nulo(l, i):
    """ó
    Funcin que devuelve el índice del primer valor de la lista l
    que es distinto de cero.

    @type l: list
    @param l: una lista de únmeros
    @type i: int
    @param i: un índice de la lista a partir del cual buscar valores no nulos
    @return: el menor índice j>=i tal que l[i]!=0, si tal índice no existe
      devuelve j=len(l)
    """
    while (i < len(l)) and (l[i] == 0):
        i += 1
    return i

def tacha(l, i, step):
    """
    Ponea cero los elementos de la lista l a partir del índice i
    saltando de step en step, es decir, pone a cero los valores
    correspondientes a los índices i+step, i+step*2, i+step*3...
    @type l: list
    @param l: lista
    @type i: int
    @param i: índice de la lista
    que esten a distancia úmltiplos de step
    @type step: int
    @param step: incremento
    """
    j = i + step
    while j < len(l):
        l[j] = 0
        j = j + step

def criba(l):
    """
    Recibe una lista de únmeros consecutivos comenzando en 2 y pone a cero
    los valores de la lista que no corresponden a únmeros primos. Utiliza
    el émtodo conocido como Criba de óEratstenes.
    @type l: list
    @param l: lista de enteros consecutivos comenzando en 2
    """
    i = 0
    while i < len(l):
        j = no_nulo(l, i)
        step = l[j]
        tacha(l, j, step)
```

```
        i = j+1

def main():
    l = range(2, 200000)
    criba(l)
    for a in l:
        if a != 0:
            print "%d," % a,
    print

if __name__=="__main__":
    main()
```

Referencias

[AH01] Jörg Ardnt and Chistoph Haenel. *Pi Unleashed*. Springer, 2001.

[BB00] Gilles Brassard and Paul Bratley. *Fundamentos de algoritmia*. Prentice-Hall, 2000.

[Bec82] Petr Beckmann. *{A history of}* π. Golem Press, 1982.

[Cha99] Jean-Luc Chabert, editor. *A History of Algorithms: From the Pebble to the Microchip*. Springer, 1999.

[Euc91] Euclides. *Elementos. Libros I-IV*. Gredos, 1991.

[Euc94] Euclides. *Elementos. Libros V-IX*. Gredos, 1994.

[Euc96] Euclides. *Elementos. Libros X-XIII*. Gredos, 1996.