Programas numéricos simples

Informática I - 2547100

Departamento de Ingeniería Electrónica
y de Telecomunicaciones
Facultad de Ingeniería

2016-2

Exhaustive enumeration

El siguiente programa calcula la raíz cúbica de un entero, si la tiene.

```
n = int(input('Enter an integer number: '))
cube = 0
while cube**3 < abs(n):
      cube = cube + 1
if cube**3 != abs(n):
      print(n, 'is not a perfect cube')
else:
      if n < 0:
            cube = -cube
      print('Cube root of', n, 'is', cube)
```

enumeración exhaustiva

for loops and range

Los ciclos **for** nos permiten expresar de una forma más simple, por ejemplo, ciclos que iteran sobre una secuencia de enteros.

```
n = int(input('Enter an integer number: '))
for cube in range(0, abs(n)+1):
        if cube **3 >= abs(n):
                break
if cube**3 != abs(n):
        print(n, 'is not a perfect cube')
else:
        if n < 0:
                cube = -cube
        print('Cube root of', n, 'is', cube)
```

```
cube = 0
while cube**3 < abs(n):
        cube = cube +1
```

for loops with strings

Los ciclos **for** también pueden iterar sobre strings...

```
#Print a string with each letter followed by a star
name = input('Enter your name: ')
for let in name:
        print(let + '*')
```

```
'''Print a string with each letter followed by a star,
but using the end configuration of print'''
name = input('Enter your name: ')
for let in name:
        print(let, end='*')
```

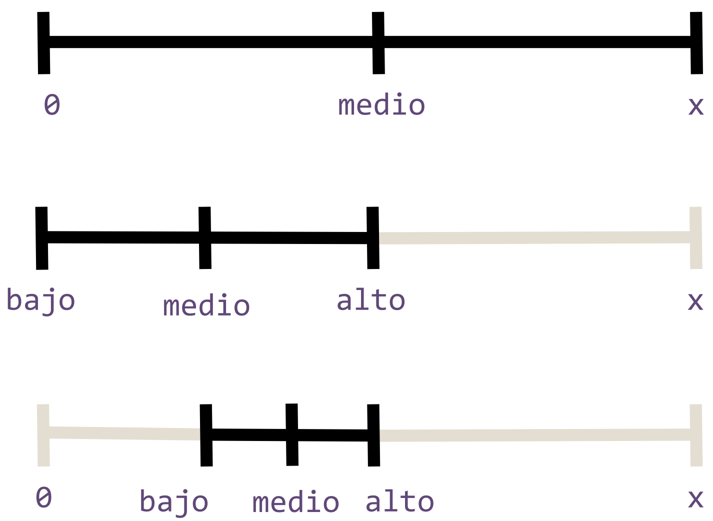
Aproximate solutions

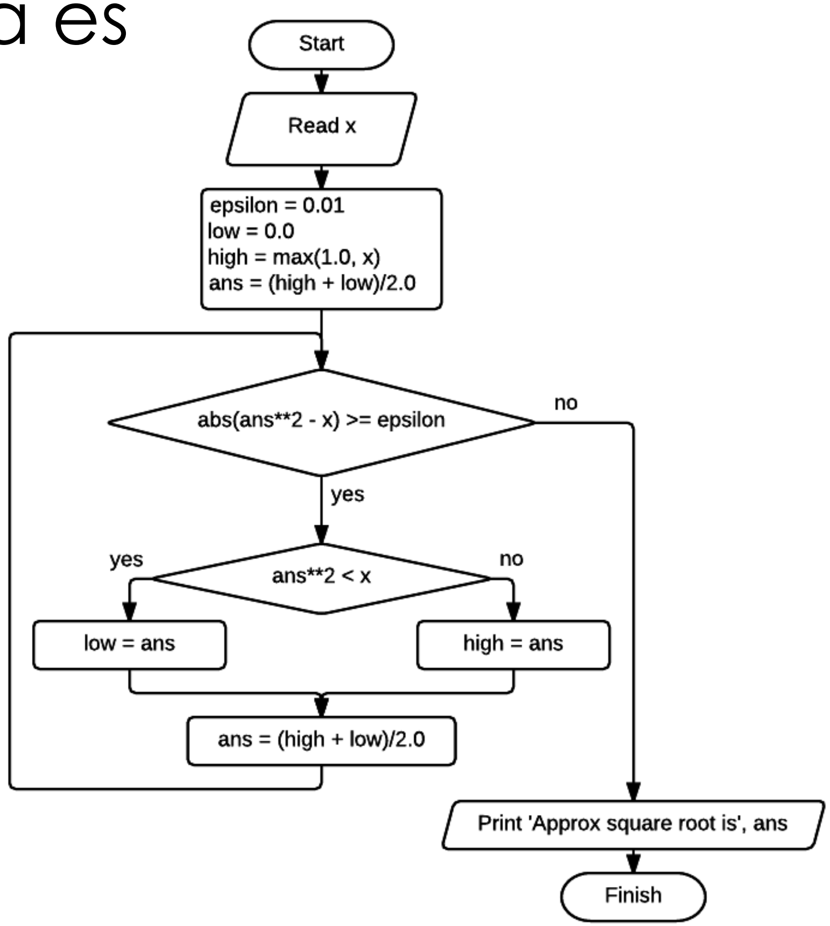
Calcular una aproximación de la raíz cuadrada de un número positivo:

```
#Find an approximation of the square root
x = 25
epsilon = 0.01
step = epsilon**2
guesses = 0
ans = 0.0
while abs(ans**2 - x) >= epsilon and ans <= x:
        ans += step
        guesses += 1
if abs(ans**2 - x) >= epsilon:
        print('Couldn\'t find the square root of', x)
else:
        print(ans, 'is approximately the square root of', x)
print('There were', guesses, 'guesses')
```

Bisection search

Para este problema, mejor que la enumeración exhaustiva es la **búsqueda binaria.**





Bisection search in Python

```
#Find a FASTER approximation of the square root
x = 25
epsilon = 0.01
guesses = 0
low = 0.0
high = max(1.0, x)
ans = (high + low)/2.0
while abs(ans**2 - x) >= epsilon:
    guesses += 1
    if ans**2 < x:
        low = ans
    else:
        high = ans
    ans = (high + low)/2.0
print(ans, 'is approximately the square root of', x)
print('There were', guesses, 'guesses')
```

Newton-Raphson

Si $\boldsymbol{g}$ es una aproximación a una raíz de un polinomio $\boldsymbol{f}$, entonces:

$$\boldsymbol{g - f(g)/f'(g)}$$

donde $\boldsymbol{f'}$ es la derivada de $\boldsymbol{f}$,es una mejor aproximación.

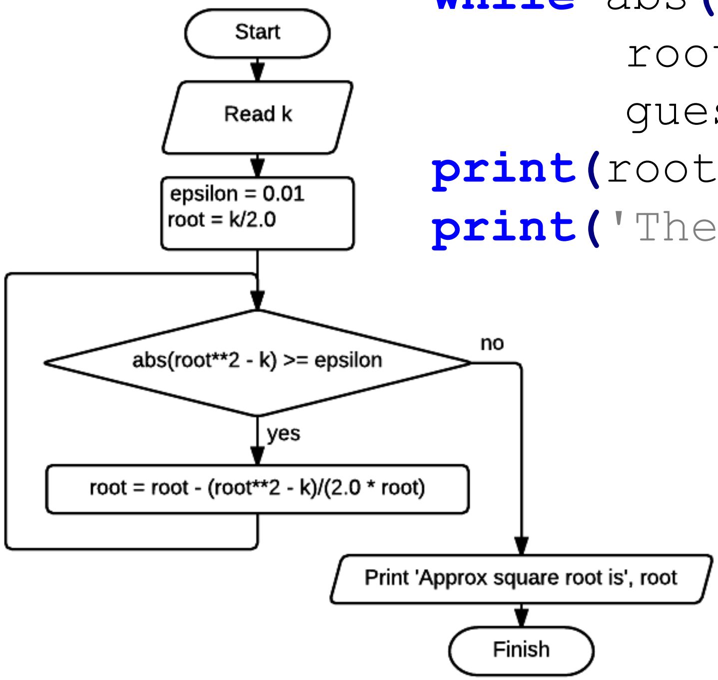
Dado que las raíces del polinomio $\boldsymbol{x^2 - k}$ nos permiten obtener la raíz cuadrada de $\boldsymbol{k}$, tenemos que la fórmula para mejorar una aproximación a la raíz cuadrada de $\boldsymbol{k}$ sería:

$$\boldsymbol{g - (g^2 - k)/2g}$$

donde $\boldsymbol{2g}$ es la derivada de $\boldsymbol{g^2 - k}$.

Newton-Raphson in Python

```python
#Using Newton-Raphson to find the square root
epsilon = 0.01
k = 24.0
root = k/2.0
guesses = 0
while abs(root**2 - k) >= epsilon:
       root = root - (root**2 - k)/(2*root)
       guesses += 1
print(root, 'is close to the square root of', k)
print('There were', guesses, 'guesses')
```



Floating-point numbers

El formato de **punto-flotante** permite representar números decimales de una manera eficiente:

```
if abs(x-1.0) < 0.0001:
```

$$3.764 \; = \; 3764 \times 10^{-3}$$

$$(\underbrace{3764}_{\textit{dígitos significativos}}, \underbrace{-3}_{\textit{exponente}})$$

En binario sería:

$$0.625 \; = \frac{5}{8} = \; 5 \times 2^{-3}$$

$$(\underbrace{101}_{\textit{bits significativos}}, \underbrace{-11}_{\textit{exponente}})$$

```
x = 0.0
for i in range(10):
        x += 0.1
if x == 1.0:
        print(x, 'is 1.0')
else:
        print(x, 'is NOT 1.0')
```

¿Cómo representar 0.1 en punto-flotante binario?

4 bits	*(0011, -101)*	*3/32*	*0.09375*
5 bits	*(11001, -1000)*	*25/256*	*0.09765625*

53 bits:
11001100110011001100110011001100110011001100110011001
0.1000000000000000055511151231257827021181583404541015625