FUNDAMENTOS DE COMPUTACIÓN
Grado en ingeniería en tecnologías Industriales
Universidad de Cantabria

Práctica 13 (laboratorio)
9 de enero del 2019

Complejidad

El objetivo de esta práctica es analizar el tiempo de ejecución de algunos programas que hemos explicado a lo largo del curso. Para ello será conveniente descargar el archivo `complejidad.py`, que contiene los programas en Python.

`http://docs.python.org/3/library/time.html`

> time.**clock**()
>
> On Unix, return the current processor time as a floating point number expressed in seconds. The precision, and in fact the very definition of the meaning of "processor time", depends on that of the C function of the same name, but in any case, this is the function to use for benchmarking Python or timing algorithms.
>
> On Windows, this function returns wall-clock seconds elapsed since the first call to this function, as a floating point number, based on the Win32 function QueryPerformanceCounter(). The resolution is typically better than one microsecond.

1. Observa el funcionamiento del método `clock` visto en el tema Algorítmica, con estos ejemplos:

```
1 from time import clock
2 def bucle(n):
3     t_0 = clock()
4     for i in range(n):
5         pass
6     t = clock()
7     return t - t_0
```

```
>>> bucle(10**6)
>>> bucle(2*10**6)
>>> bucle(3*10**6)
>>> bucle(4*10**6)
>>> bucle(10**7)
>>> bucle(10**8)
```

Como hemos visto este programa tiene complejidad lineal, es decir, $O(n)$. Si el número de iteraciones n se multiplica por un factor (digamos 100), ¿qué ocurre —*grosso modo*— con el tiempo de ejecución?

Podemos decir que la instrucción `pass` no hace nada. Se puede usar cuando una sentencia es requerida por la sintáxis pero el programa no requiere ninguna acción.

2. **Célebre anécdota escolar de Gauß.**— Construir un programa iterativo `suma1.py` para calcular la suma $\sum_{i=1}^{n} i$. Sin embargo, analizando algo más la situación, podemos conseguir un algoritmo mucho más rápido. Escribe un programa que represente en el sistema unario y por duplicado la suma $\sum_{i=1}^{n} i$, dando un resultado similar al que sigue:

```
$ python suma2_1.py
Tope: 5
 1 +  5 = @ # # # # #
 2 +  4 = @ @ # # # #
 3 +  3 = @ @ @ # # #
 4 +  2 = @ @ @ @ # #
 5 +  1 = @ @ @ @ @ #
```

Esta suma duplicada toma la forma de un rectángulo de unidades (en el ejemplo, de dimensiones 6×5). Expresa su altura y su anchura en función de n y deduce una fórmula para $2\sum_{i=1}^{n} i$.

Aprovecha la fórmula para escribir una versión ágil del programa `suma2.py` que suma los primeros n números.

Compara estas dos opciones con la función `clock`. El primer algoritmo es lineal, es decir, $O(n)$. El algoritmo de la segunda solución consiste, básicamente, en una suma, una multiplicación y una división: tres operaciones aritméticas. ¿Deducimos entonces que su tiempo de ejecución es siempre el mismo (prácticamente nada)? En otra palabras, el algoritmo tiene una complejidad constante $O(1)$. Aplícalo para valores de n muy grandes, como 10^{10^4} o 10^{10^5}.

3. Queremos comparar los programas `busqueda_lineal` de complejidad lineal $O(n)$ y `busqueda_binaria` de complejidad logarítmica $O(\log n)$, vistos en el tema ALGORITMICA

> **Entrada:** Un número a y una sucesión de n números $L = [a_1, a_2, \ldots, a_n]$.
> **Salida:** El primer índice i tal que $a_i = a$, ó -1 si a no es un elemento de L.

```
def  busqueda_lineal(a, L):
  y = -1
  for j in range(len(L)):
    if a == L[j]:
      y = j+1
      break
  return y

def  busqueda_binaria(a, L):
  menor = 0
  grande = len(L)-1
  while menor <= grande:
    mitad = (menor + grande) // 2
    if L[mitad] > a:
      grande = mitad -1
    elif L[mitad] < a:
      menor = mitad + 1
    else:
      return mitad + 1
  return -1
```

Para ello utilizamos en método `randint` del módulo RANDOM y el `clock` del TIME

```
def  tiempo_busqueda_lineal(N, M):
  t_0 = clock()
  busqueda_lineal(randint(0,10**N),range(10**M))
  t = clock()
  return t-t_0


def  tiempo_busqueda_binaria(N,M):
  t_0 = clock()
  busqueda_binaria(randint(0,10**N),range(10**M))
  t = clock()
  return t-t_0
```

Realizar pruebas para distintos valores de N y M y compara los resultados.

4. Ahora, queremos comparar el tiempo de ejecución de los tres métodos explicados en clase para ordenar una lista: MÉTODO DE LA BURBUJA, ORDENAR INSERTANDO ambos de complejidad cuadrática $O(n^2)$ y ORDENAR MEZCLANDO, de complejidad casi-lineal $O(n \log n)$.

5. Finalmente, comparamos dos formas distintas de computar el máximo común divisor de dos números: el algoritmo escolar, útil para ilustrar la definición de m.c.d., y el de Euclides. Este resulta mucho más sencillo de programar (y de llevar a la práctica en general, por tanto. ¿Estás de acuerdo?[1]).

[1] Es decir, ¿cómo es más fácil calcular máximos comunes divisores: mediante el algoritmo de Euclides o como se explica en la escuela?

euclides.py

```
def mcd(a,b):
    while b != 0:
        a, b = b, a%b
    return a
```

Con este programa podemos calcular en unos segundos el m.c.d. de números bastante grandes. ¿De qué tamaño, aproximadamente? Podemos utilizar el método `randint` contenido en el módulo `random`, que permite generar números aleatorios (visto en la práctica dedicada a los ficheros)

Vamos a modificar el programa para contar cuántas divisiones hace.

euclides.py

```
def mcd(a,b):
    contador = 0
    while b!=0:
        a,b = b, a%b
        contador += 1
    return a,contador
```

Parece factible que el ordenador haga en seguida 100 o 1 000 divisiones. Busca datos de entrada para que nuestro algoritmo tenga que llevar a cabo tantas iteraciones.

Ayuda: Usar los números de la sucesión de Fibonacci del programa anterior.

Para esos datos, ¿cómo calcularíamos el máximo común divisor según el procedimiento escolar?

El primer paso es calcular la descomposición factorial de cada uno de los dos números. Para esto podríamos utilizar el *simple* programa que construimos en clase para decidir si un numero es primo.

```
def primo(n):
    if n%2 == 1 or n == 2:
        i = 3
        while i <= int(n**(1/2)):
            if n%i == 0:
                return False
            i += 2
        return True
    return False


def primos(a,b):
    solu=[]
    if a%2==0:
        for n in range(a+1,b+1,2):
            if primo(n) == True:
                solu.append(n)
        return solu
    return primos(a-1,b)
```

Alternativamente, construimos directamente la lista de primos menores que un entero n:

```
def lista_primos(n):
    primos = [2]
    for j in range(3,n+1,2):
        divisible = False
        for p in primos:
            if j%p == 0:
                divisible = True
        if not divisible:
            primos.append(j)
    return primos
```

Puesto que nuestro programa es muy lento, quizás resulta útil contar con una tabla de números primos. En internet podemos encontrar algunas:

http://oeis.org/A000040/b000040.txt 10 000 primeros primos
http://oeis.org/A000040/a000040.txt 100 000 primeros primos

Pero podríamos necesitar números primos muy superiores a los que figuran en esas tablas. Podemos programar la tarea de construir la lista de los primos hasta una cierta cota (por ejemplo, mediante la famosa *criba de Eratóstenes*), pero ejecutar ese programa para obtener los números del tamaño que necesitamos está fuera de nuestro alcance.

Este es el algoritmo que suele aprenderse en la escuela:

escolar.py

```
# Carga los primos:
primos=[]
f=open('b000040.txt','r')
for l in f:
    aux=l.split()
    if len(aux)>1:
        primos.append(int(aux[1]))
f.close()
```

```
def mcd(a,b):
    i=0                       ###########
    desc_a=[0]                #         #
    while a>1:                #         #
        p=primos[i]           # Descomp.#
        while a%p==0:         #  de a   #
            desc_a[i]+=1      #         #
            a/=p              #         #
        i+=1                  #         #
        desc_a+=[0]           ###########
    i=0                       ###########
    desc_b=[0]                #         #
    while b>1:                #         #
        p=primos[i]           # Descomp.#
        while b%p==0:         #  de b   #
            desc_b[i]+=1      #         #
            b/=p              #         #
        i+=1                  #         #
        desc_b+=[0]           ###########
    tope=min(len(desc_a),len(desc_b))
    aux=1
    for i in range(tope):
        exponente=min(desc_a[i],desc_b[i])
        aux*=primos[i]**exponente
    return aux
```

Podemos modificarlo para que cuente cuántas divisiones realiza y compararlo (en ejemplos pequeños) con el procedimiento de Euclides.