53-COMPL-eficiencia-tiempo

December 3, 2017

Ejemplo 1

Este es el Ejercicio 4.6.3 de las notas del curso:

Define una función, de dos enteros k y a, que cuente el número de primos gemelos en cada uno de los subintervalos $[kt, k(t+1)]$ *de longitud k dentro del intervalo* $[0, ka]$.

El código procede de la respuesta a un examen de hace años.

```
In [1]: def g(m):
            P=[ x for x in xrange(m) if is_prime(x)] #Elegimos xrange(m) porque nos piden prim
            C=[]

            for n in srange(len(P)):              #Para cada elemento de la lista de primos men
                if n==len(P)-1:                  #Si ha llegado al último elemento, se para
                    break
                elif P[n+1]-P[n]==2:             #Si no, comprobamos si ese primo y el siguien
                    C = C + [[P[n],P[n+1]]]      #Añadimos la pareja a la lista C que nos devo

            return C

In [2]: def g2(k,a):
            G=[]
            for t in xrange(a):
                I= [k*t, k*(t+1)]
                for x in xrange(len(g(k*a+1))):
                    if (I[0]<=g(k*a+1)[x][0]) and (I[1]>=g(k*a+1)[x][1]):
                        G=G+[I]
            return G

In [3]: def g3(k,a):
            G=g2(k,a)
            for t in srange(a):
                I= [k*t, k*(t+1)]
                if G.count([k*t, k*(t+1)])==0:
                    print 'No hay parejas de primos gemelos en',[k*t, k*(t+1)]
                elif G.count([k*t, k*(t+1)])==1:
                    print 'Hay',G.count([k*t, k*(t+1)]),'pareja de gemelos en',[k*t, k*(t+1)]
                else:
                    print 'Hay',G.count([k*t, k*(t+1)]),'parejas de gemelos en',[k*t, k*(t+1)]
```

```
In [4]: %time g3(1000,10)

Hay 35 parejas de gemelos en [0, 1000]
Hay 26 parejas de gemelos en [1000, 2000]
Hay 20 parejas de gemelos en [2000, 3000]
Hay 21 parejas de gemelos en [3000, 4000]
Hay 23 parejas de gemelos en [4000, 5000]
Hay 17 parejas de gemelos en [5000, 6000]
Hay 19 parejas de gemelos en [6000, 7000]
Hay 13 parejas de gemelos en [7000, 8000]
Hay 14 parejas de gemelos en [8000, 9000]
Hay 15 parejas de gemelos en [9000, 10000]
CPU times: user 1min 16s, sys: 132 ms, total: 1min 16s
Wall time: 1min 16s


In [5]: %prun -q -T profiler1.txt g3(1000,10)

Hay 35 parejas de gemelos en [0, 1000]
Hay 26 parejas de gemelos en [1000, 2000]
Hay 20 parejas de gemelos en [2000, 3000]
Hay 21 parejas de gemelos en [3000, 4000]
Hay 23 parejas de gemelos en [4000, 5000]
Hay 17 parejas de gemelos en [5000, 6000]
Hay 19 parejas de gemelos en [6000, 7000]
Hay 13 parejas de gemelos en [7000, 8000]
Hay 14 parejas de gemelos en [8000, 9000]
Hay 15 parejas de gemelos en [9000, 10000]

*** Profile printout saved to text file u'profiler1.txt'.
```

Vemos que se ha ejecutado unas 30 millones de veces la instrucción is_prime, y eso se ha llevado casi todo el tiempo de ejecución.£Hace falta realmente ejecutar 30 millones de veces is_prime? Como todas las comprobaciones ocurren en el intervalo $[1,10000]$ perece realmente exagerado. Por otra parte, el programa completo contiene un montón de bucles (he contado 5), pero lo peor es que dentro de g2 llama un montón de veces a g (vemos que son 3033 veces) y en cada una de esas llamadas ejecuta un montón de is_prime. Es eso lo que hace que se ejecute is_prime 30 millones de veces.

£Cuántas veces es razonable que se ejecute is_prime cuando calculamos en el intervalo $[1,10000]$? Podrían ser 10000 veces, para comprobar si uno de los enteros del intervalo es primo, más una vez más cada vez que encontramos un primo para comprobar si sumándole 2 sigue siendo primo. En cualquier caso bastante menos de 20000 llamadas en total.

Comprobemos con un programa mucho más sencillo que calcula la misma cosa:

```
In [6]: def numero_primos(x0,x1):
            cont = 0
            for k in srange(x0,x1):
                if is_prime(k) and is_prime(k+2):
```

```
                    cont += 1 ##Si se cumplen las condiciones de primos gemelos incrementamos
            return cont
        def lista_num_gem(k,a):
            return [numero_primos(k*t,k*(t+1)) for t in srange(0,a)]
```

```
In [7]: %time lista_num_gem(1000,10)

CPU times: user 8 ms, sys: 4 ms, total: 12 ms
Wall time: 7.71 ms


Out[7]: [35, 26, 21, 21, 23, 17, 19, 13, 15, 15]

In [8]: %prun -q -T profiler2.txt lista_num_gem(1000,10)


*** Profile printout saved to text file u'profiler2.txt'.
```

En resumen, aunque los dos programas producen esencialmente el mismo resultado y deben ser ambos correctos, el segundo llama 11200 veces a is_prime mientras que el primero lo hace 30 millones de veces. La lógica correcta para este programa consiste en:

Una función que cuente (usando un contador) el número de primos gemelos en un intervalo fijado mediante un bucle for, ejecutado entre los extremos del intervalo dado, y un if para incrementar el contador.

Una función que ejecute, mediante otro for esta vez en a, el primer programa para cada uno de los subintervalos que nos piden.

En total el "número de vueltas" que se ejecutan para los dos for es, en nuestro ejemplo, 10×1000 .

Ejemplo 2

Es el Ejercicio 4.6.5 de las notas del curso:

Dado un entero N define una función de N que devuelva el subintervalo $[a,b]$ de $[1,N]$ más largo tal que no contenga ningún número primo.

El código también procede de la respuesta a un examen de hace años.

```
In [9]: def subintervalo(N):
            P=[s for s in xrange(N) if is_prime(s)]
            if is_prime(N):
                P=[0]+P+[N]
            else:
                P=[0]+P+[N+1]           #El intervalo [1,N] va a quedar dividido por estos núme
            I=[]
            for n in srange(1,len(P)):
                I=I+[[P[n]-P[n-1]-2,[P[n-1]+1,P[n]-1]]] #Creamos la lista I, que nos da los ex
                                                        #precedidos por la amplitud de cada in
            I.sort()                                    #Ordenamos I para ver cual es la mayor
            I.reverse()                                 #La invertimos para tener las mayores
            C=[I[0][1]]                                 #Y tomamos C como el primero de los in
            for n in srange(1,len(I)):                  #Y vemos si hay más de un intervalo co
```

```
            if C[0][1]-C[0][0]==I[n][0]:            #En cuyo caso, lo añadimos a C
                C=C+[I[n][1]]
        return C
```

```
In [10]: time subintervalo(1000000)
```

```
CPU times: user 24.5 s, sys: 352 ms, total: 24.9 s
Wall time: 24.6 s
```

```
Out[10]: [[492114, 492226]]
```

```
In [11]: %prun -q -T profiler3.txt subintervalo(1000000)
```

```
*** Profile printout saved to text file u'profiler3.txt'.
```

Esto nos dice que casi todo el tiempo se gasta en crear la lista P y en recorrerla para buscar la mayor diferencia entre dos enteros consecutivos de ella. Como sólo aparecen las llamadas a funciones sólo vemos el tiempo gastado en *is_prime*.

Lo que parece ineficiente son los dos bucles for y la misma idea de crear una lista I que en realidad NO hace falta.

En las dos celdas que siguen se obtiene la misma lista de primos con prime_range y mediante la misma instrucción que se usa dentro de subin: vemos que la función prime_range tarda muy poco £por qué?

```
In [12]: time L = prime_range(1000000)
```

```
CPU times: user 8 ms, sys: 0 ns, total: 8 ms
Wall time: 6.79 ms
```

```
In [13]: time L = [n for n in xrange(1000000) if is_prime(n)]
```

```
CPU times: user 2.73 s, sys: 380 ms, total: 3.11 s
Wall time: 2.73 s
```

```
In [14]: ## prime_range??
```

```
In [15]: ## next_prime??
```

Si podemos usar next_prime el código es mucho más eficiente:

```
In [1]: def subintervalo2(N):
            p = 2
            q = 3
            maximo = [2,3,1]
            while next_prime(q) < N:
                p,q = q,next_prime(q)
                if q-p > maximo[2]:
                    maximo = [p,q,q-p]
            return maximo
```

```
In [2]: time subintervalo2(1000000)

CPU times: user 704 ms, sys: 64 ms, total: 768 ms
Wall time: 686 ms


Out[2]: [492113, 492227, 114]

In [3]: %prun -q -T profiler4.txt subintervalo2(1000000)


*** Profile printout saved to text file u'profiler4.txt'.


In [4]: def subintervalo3(N):
            L=[n for n in xrange(N) if is_prime(n)]
            return max([L[i+1]-L[i] for i in xrange(0,len(L)-1)])

In [5]: time subintervalo3(1000000)

CPU times: user 2.96 s, sys: 128 ms, total: 3.09 s
Wall time: 2.95 s


Out[5]: 114

In [6]: %prun -q -T profiler5.txt subintervalo3(1000000)


*** Profile printout saved to text file u'profiler5.txt'.
```

La función subintervalo3 parece lo mejor que se puede hacer si usamos llamadas a is_prime en lugar de a next_prime, y debería considerarse la versión correcta del código subin. No es correcto comparar subin con subintervalo2, ya que subin usa is_prime y lo llama 1000000 de veces y subintervalo2 usa next_prime y lo llama sólo 157000 veces.

La comparación entre subin y subintervalo3 muestra que subin gasta aproximadamente 6/7 partes de su tiempo procesando de manera muy ineficiente la lista P.

EN RESUMEN:

Hay que intentar crear códigos con el menor número posible de bucles (for o while). Sobre todo hay que evitar bucles innecesarios.

El "profiler" (cProfile en nuestro caso) nos indica cuánto tiempo gasta el programa ejecutando llamadas a funciones, como is_prime, y coloca arriba de la lista las que más tiempo consumen. Si las hemos programado nosotros esas son las que debemos mejorar y si son funciones de SAGE podemos pensar si habrá otra más eficiente en nuestro problema (por ejemplo, usar next_prime en lugar de is_prime) o tratar de reducir el número de llamadas a esas funciones reduciendo, si es posible, el número de bucles o usando bucles con menos vueltas.