Técnicas de programación y complejidad de algoritmos

Hay varias formas de definir el costo computacional de un algoritmo. Una primera forma es simplemente contar la cantidad de operaciones que el algoritmo hace durante su ejecución. A eso se lo llama la *complejidad aritmética* del algoritmo. Esta complejidad es independiente de la máquina en la que el algoritmo se ejecute y da una cota inferior del verdadero tiempo de cálculo. Es claro que las operaciones aritméticas (suma, resta, multiplicación, división) de enteros son mas costosas para números largos que para números cortos. La complejidad aritmética no tiene en cuenta esas diferencias. Lo mismo sucede con la concatenación de listas y tuplas, la unión e intersección de conjuntos, etc. Es por eso que se introduce tambień la noción de *complejidad binaria*, que cuenta la cantidad de operaciones que el procesador de la máquina hará durante la ejecución del algoritmo. El problema con esta definición es que se necesita especificar el tipo de máquina se utilizará y también cómo las instrucciones de Python3 se traducen en operaciones de la máquina.

Empecemos con un ejemplo simple de algoritmo que dado un entero x calcula su raíz cuadrada $\lfloor\sqrt{x}\rfloor$ redondeada hacia abajo. Es decir, busca el mayor entero y tal $y^2 \geq x$.

```
1 def raiz_cuadrada_muy_lenta(x):  # x >= 0
2     y = x
3     while y*y > x:
4         y -= 1
5     return y
```

Programa 1: `raizcuad.py` (lineas 1–5)

Es fácil ver que este algoritmo es correcto para cualquier entrada $x \geq 0$. Es importante indicar en la especificación del algoritmo que este nunca debe ser invocado con un argumento negativo.

¿Cuánto tiempo, medido en número de operaciones aritméticas entre enteros y comparaciones, tarda el algoritmo `raiz_cuadrada_muy_lenta` en terminar? Un conteo rápido nos dice que la asignación de la linea 2 se ejecuta una sola vez, la multiplicación y comparación de la linea 3 se ejecutan $x - \lfloor\sqrt{x}\rfloor + 1$ veces y la resta de la linea 4 se ejecuta $x - \lfloor\sqrt{x}\rfloor$ veces. En total, el algoritmo realiza $3(x - \lfloor\sqrt{x}\rfloor + 1)$ operaciones. Esto es la complejidad aritmética del algoritmo.

Sean $f, g : \mathbb{N} \to \mathbb{R}_{>0}$ y supongamos que g es creciente. Decimos que

$$f \in O(g) \iff \exists\, c, x_0 > 0 \;:\; \forall\, x > x_0 \;:\; f(x) \leq cg(x) \iff \limsup_{x\to\infty} \frac{f(x)}{g(x)} < \infty$$

Por ejemplo, se puede decir que la complejidad aritmética de `raiz_cuadrada(x)` es $O(x)$. Lógicamente, también se podría decir que es $O(x^2)$, pero es mejor la anterior, ya que da una cota mas estricta.

En análisis de algoritmos es mas común expresar la complejidad de los algoritmos en función del *tamaño* de la entrada y no en función del valor numérico de la entrada. Si consideramos que el número x está expresado en base 10, su tamaño será aproximadamente $n = size(x) \approx \log_{10}|x|$, por lo que la complejidad aritmética de `raiz_cuadrada(x)` sería $\approx 3 \cdot 10^n \in O(10^n)$, es decir, exponencial en el tamaño de la entrada.

Se puede conseguir una mejora importante en el algoritmo anterior si en lugar de empezar a explorar los posibles y empezando en x lo hicieramos empezando en 1. De ese modo conseguiremos la solución en $O(\sqrt{x})$ pasos, es decir, con complejidad aritmética $O(10^{n/2})$.

```
7  def raiz_cuadrada_lenta(x):       # x >= 0
8      y = 1
9      while y*y <= x:
10         y += 1
11     return y-1
```

Programa 2: `raizcuad.py` (lineas 7–11)

Una idea bastante mejor es hacer una *busqueda binaria* de y en el intervalo $[0, x]$, aprovechando que los cuadrados perfectos están ordenados de forma creciente $0^2 < 1^2 < 2^2 < 3^2 < \cdots < x^2$.

```
13 def raiz_cuadrada(x):             # x >= 0
14     izq = 0
15     der = x+1
16     while izq < der-1:            # buscamos en [izq,der)
17         med = (izq+der)//2
18         if med*med <= x:
19             izq = med
20         else:
21             der = med
22     return izq
```

Programa 3: `raizcuad.py` (lineas 13–22)

En cada iteración, la longitud del intervalo $[izq, der)$ se reduce a la mitad, por lo que tendremos $\approx \log_2(x) = \log_{10}(x)/\log_{10}(2)$ pasos hasta salir del bucle. Esto nos dice que la complejidad aritmética del algoritmo `raiz_cuadrada_super_rapida(x)` es $O(n)$, donde $n = size(x) \approx \log_{10}(x)$.

Ahora veamos como calcular el máximo común divisor de dos enteros $(x, y) \neq (0, 0)$ dados. La idea más simple es explorar todos los enteros en $[1, \max\{|x|, |y|\}]$, de mayor a menor y detenerse cuando aparezca el primero que divida a x y a y.

```
1  def gcd_lento(x,y):                # (x,y) != (0,0)
2      m = min(abs(x), abs(y))
3      while x%m != 0 or y%m != 0:
4          m -= 1
5      return m
```

Programa 4: `gcds.py` (lineas 1–5)

Por supuesto, ya hemos visto que ese tipo de búsqueda son extremadamente ineficientes con respecto a la cantidad de operaciones que deben realizarse. De hecho, si llamamos $n = size(x) + size(y)$, resulta que la complejidad aritmética de `gcd_lento(x,y)` es $O(10^n)$.

Para conseguir una complejidad razonable, necesitamos un algoritmo que reduzca la cantidad de dígitos en cada iteración. Una posibilidad es la siguiente: si x e y son ambos pares, entonces claramente $\gcd(x, y) = 2 \cdot \gcd(x/2, y/2)$. Por otra parte, si x es par pero y es impar, tenemos que $\gcd(x, y) = \gcd(x/2, y)$. De forma similar, si x es impar e y es par, tenemos $\gcd(x, y) = \gcd(x, y/2)$. Estas identidades nos permiten dividir por 2 a al menos uno de los datos de entrada. Hasta ahí todo bien, pero ¿qué hacer si x e y son impares? Un posible truco es que $\gcd(x, y) = \gcd(x, y - x)$ y ahora, como $y - x$ es par, en el siguiente paso tendremos $\gcd(x, y) = \gcd(x, (y - x)/2)$. Para cualquier $x, y \geq 0$ (no ambos nulos) tenemos la siguiente fórmula para calcular $\gcd(x, y)$:

$$\gcd(x,y) = \begin{cases} y & \text{si } x = 0 \\ x & \text{si } y = 0 \\ 2\gcd(\frac{x}{2}, \frac{y}{2}) & \text{si } x, y \text{ son ambos pares} \\ \gcd(\frac{x}{2}, y) & \text{si } x \text{ es par, } y \text{ es impar} \\ \gcd(x, \frac{y}{2}) & \text{si } x \text{ es impar, } y \text{ es par} \\ \gcd(\frac{x-y}{2}, y) & \text{si } x, y \text{ son ambos impares con } x > y \\ \gcd(x, \frac{y-x}{2}) & \text{en caso contrario} \end{cases}$$

```
7  def gcd_binario(x,y):                # (x,y) != (0,0)
8      x = abs(x)
9      y = abs(y)
10     xespar = x%2 == 0
11     yespar = y%2 == 0
12     if x == 0:                       # caso base: gcd(0,y)=y
13         m = y
14     elif y == 0:                     # caso base: gcd(x,0)=x
15         m = x
16     elif xespar and yespar:
17         m = 2 * gcd_binario(x//2, y//2)
18     elif xespar:
19         m = gcd_binario(x//2, y)
20     elif yespar:
21         m = gcd_binario(x, y//2)
22     elif x > y:
23         m = gcd_binario(y, (x-y)//2)
24     else:
25         m = gcd_binario(x, (y-x)//2)
26     return m
```

Programa 5: `gcds.py` (lineas 7–26)

Es fácil comprobar que `gcd_binario(x,y)` tiene una complejidad aritmética $O(n)$, donde $n = size(x)+size(y)$. En cada paso, uno de los números pierde (al menos) 1 dígito binario de longitud.

Un algoritmo que se invoca a si mismo se llama recursivo. La idea es que en cada llamada el problema se va reduciendo a un caso cada vez mas fácil, hasta en algún momento dar con los *casos base* que se resuelven directamente.

Consideremos ahora la sucesión de Fibonacci, definida recursivamente del siguiente modo:

$$\begin{aligned} f_0 &= 0, \\ f_1 &= 1, \\ f_n &= f_{n-1} + f_{n-2} \quad \text{si } n \geq 2. \end{aligned}$$

Esa definición puede traducirse directamente en la función `fibonacci_rec(n)` que calcula f_n.

```
1  def fibonacci_rec(n):    # n >= 0
2      if n == 0:
3          return 0
4      elif  n == 1:
5          return 1
6      else:
7          return fibonacci_rec(n-1) + fibonacci_rec(n-2)
```

Programa 6: `fibo.py` (lineas 1–7)

El problema con la implementación anterior es que es extremadamente lenta. Se puede ver que la llamada `fibonacci_rec(30)` se convierte en las llamadas `fibonacci_rec(29)` y `fibonacci_rec(28)`, y cada una de esas de convierte en dos llamadas y así siguiendo. Al final, se terminarán haciendo $f_{30} = 1346269$ llamadas a los casos base `fibonacci_rec(0)` y `fibonacci_rec(1)`.

Es claro que el problema anterior proviene de que la función `fibonacci_rec(n)` no tiene "memoria". Eso se puede solucionar de dos formas diferentes. La primera de estas, que se llama *memoización*, consiste en almacenar los resultados de cada invocación en memoria (usualmente en un diccionario o una lista), y cada vez que se recibe una llamada, se busca primero en memoria si el resultado ya está disponible o si debe ser calculado.

```
9  resultados_aprendidos = {}    # diccionario vacio
10
11 def fibonacci_mem(n):         # n >= 0
12     if n in resultados_aprendidos:
13         return resultados_aprendidos[n]
14     else:
15         if n == 0:
16             resultado = 0
17         elif n == 1:
18             resultado = 1
19         else:
20             resultado = fibonacci_mem(n-1) + fibonacci_mem(n-2)
21         resultados_aprendidos[n] = resultado
22         return resultado
```

Programa 7: `fibo.py` (lineas 9–22)

La función `fibonacci_mem(n)` permite calcular el valor de f_n en solamente $O(n)$ operaciones aritméticas. La única limitación de esta versión estará determinada por la máxima profundidad recursiva que el intérprete de Python3 permita (típicamente alrededor de 1000). Por ejemplo, la llamada `fibonacci_mem(10000)` dará error en casi todas las implementaciones modernas de Python3.

Detalle técnico: Si bien no es recomendado, se puede cambiar la máxima profundidad de la recursión con el comando `sys.setrecursionlimit(n)` de la librería estándar `sys`.

Otra posible técnica, es eliminar la recursión completamente y reescribir el algoritmo de forma iterativa. Para esto, es importante identificar la cadena de dependencias entre los subproblemas, e ir resolviendolos en orden creciente de dificultad. Los resultados de las iteraciones se van almacenando en una estructura de datos, de forma similar a la memoización. En cada iteración solo se necesitan valores ya calculados previamente.

```
24 def fibonacci_ite(n):         # n >= 0
25     fib = [0] * (n+1)         # reservamos espacio para f0,f1,...,fn
26     fib[0] = 0
27     fib[1] = 1
28     for i in range(2,n+1):
29         fib[i] = fib[i-1] + fib[i-2]
30     return fib[n]
```

Programa 8: `fibo.py` (lineas 24–30)

Esta implementación también tiene complejidad aritmética $O(n)$, pero no recuerda los valores para invocaciones siguientes. Por otra parte, como no utiliza recursión, no hay otro límite más que la capacidad de la memoria en donde se almacenan los resultados previos. Ambas técnicas se llaman *programación dinámica* y es perfectamente posible usarlas de forma combinada.

Volvamos a analizar la implementación de `gcd_binario(x,y)`. Se puede ver que cada uno de los `return` termina con el caso base o con una llamada a la misma función. Esto puede traer problemas con el límite que impone Python3 sobre la profundidad de la recursión. Sin embargo, es posible en este caso utilizar un truco llamado *tail recursion* que elimina completamente la recursión. La idea es reducirse primero al caso base utilizando un bucle while y luego devolver lo que corresponda según el caso al que se llegue.

```
28 def gcd_binario_tail_rec(x,y):     # (x,y) != (0,0)
29     x = abs(x)
30     y = abs(y)
31     m = 1
```

```
32      while x != 0 and y != 0:
33          xespar = x%2 == 0
34          yespar = y%2 == 0
35          if xespar and yespar:
36              m *= 2                  # aquí acumulamos las potencias de 2
37              x //= 2
38              y //= 2
39          elif xespar:
40              x //= 2
41          elif yespar:
42              y //= 2
43          elif x > y:
44              x = (x-y)//2
45          else:
46              y = (y-x)//2
47      if x == 0:                      # caso base: gcd(0,y)=y
48          m *= y
49      else:                           # caso base: gcd(x,0)=x
50          m *= x
51      return m
```

Programa 9: `gcds.py` (lineas 28–51)

Por último, acabamos este tema con una implementación mas fancy (y algo mas rápida) del algoritmo `gcd_binario_tail_rec` que utiliza los operadores de manipulación de bits.

```
53  def gcd_binario_tail_rec_fancy(x,y):    # (x,y) != (0,0)
54      x = abs(x)
55      y = abs(y)
56      s = 0
57      while x != 0 and y != 0:
58          xespar = x&1 == 0
59          yespar = y&1 == 0
60          if xespar and yespar:
61              s += 1                  #  m = 2**s = 1 << s
62              x >>= 1
63              y >>= 1
64          elif xespar:
65              x >>= 1
66          elif yespar:
67              y >>= 1
68          elif x > y:
69              x = (x-y) >> 1
70          else:
71              y = (y-x) >> 1
72      if x == 0:                      # caso base: gcd(0,y)=y
73          m = y << s
74      else:                           # caso base: gcd(x,0)=x
75          m = x << s
76      return m
```

Programa 10: `gcds.py` (lineas 53–76)

Problema 2.1. Demostrar que para cualquier par de enteros $(x, y) \neq (0, 0)$, existen $a, b \in \mathbb{Z}$ tales que $ax + by = \gcd(x, y)$. Implementar una función `gcd_extendido_binario(x,y)`, basado en la idea de `gcd_binario`, que no solo devuelva el máximo común divisor, sino que también devuelva a y b.

Problema 2.2. El método clásico para calcular el máximo común divisor se llama *algoritmo de Euclides* y está basado en la identidad $gcd(x, y) = gcd(y, x \bmod y)$, donde $x \bmod y$ es el resto de dividir

x por y. Demostrar que la identidad es correcta, e implementar el correspondiente algoritmo recursivo `gcd_euclides(x,y)` identificando cuáles son los casos base. Comparar la complejidad aritmética de este algoritmo con `gcd_binario(x,y)`. Por último, extender el algoritmo a uno que obtenga también a y b como en el problema 2.1

Problema 2.3. Implementar una función `es_suma_de_k_potencias_n(x,k,n)` que decida si un entero $x \geq 0$ se puede escribir como $x_1^n + x_2^n + \cdots + x_k^n$ para ciertos enteros $x_1, \ldots, x_k \geq 0$. La función debe devolver un valor booleano.

Problema 2.4. Considerar el siguiente juego de dos jugadores: partiendo de una pila de n piedras, los jugadores van (uno tras otro) quitando 1, 2, 6 o 98 piedras de la pila a su elección, hasta que el que quita la última pierde. Implementar un algoritmo recursivo `hay_estrategia_ganadora(n)` que determine si, partiendo de una pila de n piedras, el jugador que empieza tiene estrategia ganadora.

Problema 2.5. Implementar una función `contar_palabras(n)` que calcule cuántas palabras hay de longitud $n \geq 1$ que utilizan solamente las letras a y b y que no tienen tres a consecutivas.

Problema 2.6 (Egg dropping problem)**.** Consideremos un edificio de $n \geq 1$ plantas y $k \geq 1$ huevos ideales con la siguiente propiedad: si se deja caer uno de estos huevos desde la planta m o superior, el huevo se rompe, pero si se lo deja caer desde una planta inferior a la m, el huevo queda intacto. El valor de $m \in [1, n+1]$ es el mismo para todos los huevos. Se pretende determinar el valor exacto de m utilizando los k huevos. Si se utiliza la estrategia óptima, ¿cuál es la mínima cantidad de veces que debe lanzar el huevo para conseguir esto? Implementar la función `egg_drop(n,k)` que lo determine. Por ejemplo, si solo disponemos de $k = 1$ huevo, la única opción es lanzarlo desde todos los pisos, empezando desde mas abajo. La primer planta desde la que se rompa, será el valor de m buscado. Si no se rompe desde ninguna planta, será $m = n + 1$. En el peor caso, necesitariamos n lanzamientos, y por lo tanto `egg_drop(n,1)` devolverá n.

Caso particular de prueba: `egg_drop(100,2)=14`.

Problema 2.7. La forma más simple de multiplicar dos matrices A de $m \times n$ y B de $n \times k$ requiere hacer $m \cdot n \cdot k$ multiplicaciones de números y una cantidad similar de sumas. Si se tienen varias matrices $A_1, A_2, \ldots, A_k$ de tamaños $n_0 \times n_1, n_1 \times n_2, \ldots, n_{k-1} \times n_k$, el producto $A_1 \cdot A_2 \cdots A_k$ puede calcularse de varios modos (por la propiedad asociativa del producto de matrices), pero no siempre con el mismo coste computacional. Implementar la función `optimal_mat_mul(tam)` que tome una tupla de enteros positivos `tam=(n0, n1, ..., nk)` y determine cuantas operaciones de multiplicación requiere la estrategia óptima para multiplicar las matrices.

Caso particular de prueba: `optimal_mat_mul((a,b,c,d))` dará el mínimo entre $abc + acd$ y $bcd + abd$. El primer valor corresponde con el coste de hacer $(A_0 \cdot A_1) \cdot A_2$ y el otro el de $A_0 \cdot (A_1 \cdot A_2)$.

Problema 2.8 (Pots of Gold)**.** En este famoso juego, dos jugadores van alternadamente llevandose pilas de oro, con el objetivo de obtener al final la mayor cantidad de oro posible. Las pilas de oro están dispuestas en una fila horizontal, y los jugadores en su turno solo pueden elegir de entre las dos pilas que están en los extremos de la fila. Digamos que hay $k \geq 1$ pilas de oro y que las cantidades en gramos de oro de cada pila están en una lista `l = [l1, l2, ..., lk]` de enteros. Escribir una función `max_gold(l)` que devuelva la máxima cantidad de oro que puede llevarse el jugador que empieza si este utiliza la estrategia óptima.

Problema 2.9. Se tiene un tablero de $n \times m$ con un número entero en cada casilla. Una ficha se encuentra inicialmente en la casilla de arriba a la izquierda del tablero. En cada movimiento se puede desplazar a esta ficha una casilla hacia abajo o una hacia la derecha. El objetivo es llegar a la casilla de abajo a la derecha de modo que la suma de los números de las casillas visitadas sea mínimo. Implementar una función `min_suma_casillas(n,m,a)` que tome una matriz `a` (una lista de n listas de m enteros) y que devuelva la suma mínima de los números visitados en el camino óptimo.