Aritmética de enteros largos

Si bien Python3 es un lenguaje de alto nivel que permite trabajar con enteros de cualquier cantidad de dígitos, y por lo tanto no necesitamos preocuparnos en implementar las operaciones aritméticas en términos de los dígitos, es interesante ver como estas podrían ser implementadas si eso no fuera así.

A la cantidad de operaciones elementales con dígitos (o más precisamente, enteros cortos con los que el procesador puede trabajar) que requiere un algoritmo se la llama *complejidad binaria*. Esta noción corresponde con el tiempo de ejecución de un algoritmo de forma independiente de la máquina en la que el algoritmo se ejecute.

Por simplicidad, vamos a implementar solamente la suma, resta, multiplicación, división (con resto) y comparación de números enteros no negativos. Utilizaremos la siguiente representación: un número $n \geq 0$ se corresponderá con la lista `[d_0,d_1,...,d_k]` donde `d_0` es el dígito de las unidades, `d_1` las decenas, y así siguiendo. Supondremos que `d_k` es siempre no nulo. El número 0 se representará con la lista vacía `[]`.

A continuación vamos a ver como implementar los algoritmos de la escuela para la suma, resta, comparación y multiplicación de números naturales. Vamos a poner todas la funciones en el fichero `natural.py` para luego poder utilizarlas en otros programas.

La función `remover_ceros(a)` toma una lista a de dígitos y le elimina los ceros a la izquierda, es decir, la reduce a una representación válida de acuerdo con nuestra convención.

```
1  def remover_ceros(a):                # a = lista de digitos decimales
2      n = len(a)
3      while n >= 1 and a[n-1] == 0:
4          n -= 1
5      del a[n:]
```

Programa 1: `natural.py` (lineas 1–5)

El algoritmo de la suma se implementa dígito a dígito, empezando por las unidades.

```
7  def sumar(a, b):                 # a,b son listas de digitos "reducidas"
8      n = len(a)
9      m = len(b)
10     if n < m:                    # nos aseguramos que a sea el mas largo
11         b, a = a, b
12         n, m = m, n
13     c = [0] * (n+1)              # reservamos espacio suficiente para la suma
14     x = 0                        # x es el acarreo, que inicialmente es 0
15     i = 0
16     while i < m:                 # i=0,1,...,m-1
17         x = a[i] + b[i] + x
18         c[i] = x % 10
19         x //= 10
20         i += 1
21     while i < n:                 # i=m,m+1,...,n-1
22         x = a[i] + x
23         c[i] = x % 10
24         x //= 10
25         i += 1
26     c[n] = x                     # guardamos el ultimo acarreo
```

```
27     remover_ceros(c)          # eliminamos los ceros a la izquierda
28     return c
```

Programa 2: `natural.py` (lineas 7–28)

De forma similar, es fácil implementar la comparación `comparar(a,b)` que devuelve $1, 0, -1$ dependiendo si $a > b$, $a = b$ o $a < b$, y la resta `restar(a,b)` que calcula $a - b$, suponiendo que $a \geq b$. La resta se hace empezando por las unidades, pero la comparación empieza por el dígito más significativo. Cuando la resta es invocada con $a < b$, la función no devuelve nada.

```
30 def comparar(a, b):                # a,b son listas de digitos "reducidas"
31     n = len(a)
32     m = len(b)
33     # si a y b son de distintas longitudes, la comparacion es inmediata,
34     # ya que la representacion que usamos no tiene ceros a la izquierda
35     if n > m:
36         return 1
37     if n < m:
38         return -1
39     # buscamos el menor indice n a partir del cual a y b coinciden
40     while n >= 1 and a[n-1] == b[n-1]:
41         n -= 1
42     if n == 0:
43         return 0
44     elif a[n-1] > b[n-1]:
45         return 1
46     else:
47         return -1
```

Programa 3: `natural.py` (lineas 30–47)

```
49 def restar(a, b):             # a,b son listas de digitos "reducidas"
50     n = len(a)
51     m = len(b)
52     # si se nos asegura que la funcion es llamada con a >= b, las
53     # siguientes dos lineas son innecesarias
54     if n < m:
55         return
56     c = [0] * n               # crear c = lista de n ceros
57     x = 0                     # inicializar el acarreo x a cero
58     i = 0
59     while i < m:              # i=0,1,...,m-1
60         x = a[i] - b[i] + x
61         c[i] = x % 10
62         x //= 10
63         i += 1
64     while i < n:              # i=m,m+1,...,n-1
65         x = a[i] + x
66         c[i] = x % 10
67         x //= 10
68         i += 1
69     # al igual que dijimos al principio, las siguientes dos lineas
70     # son innecesarias si a >= b.
71     if x != 0:
72         return
73     remover_ceros(c)
74     return c
```

Programa 4: `natural.py` (lineas 49–74)

Las funciones `sumar(a,b)`, `comparar(a,b)` y `restar(a,b)` reciben una entrada de tamaño $n = len(a) + len(b)$ y se puede comprobar que sólo requieren $O(n)$ operaciones entre dígitos.

La multiplicación mediante el algoritmo de la escuela es también fácil de implementar, pero luego veremos que no es el método más apropiado para números muy largos.

```python
76 def multiplicar(a, b):          # a,b son listas de digitos "reducidas"
77     n = len(a)
78     m = len(b)
79     c = [0] * (n+m)
80     for i in range(n):                                  # i = 0,1,...,n-1
81         x = 0
82         for j in range(m):                              # j = 0,1,...,m-1
83             x = c[i+j] + a[i] * b[j] + x
84             c[i+j] = x % 10
85             x //= 10
86         c[i+m] = x
87     remover_ceros(c)
88     return c
```

Programa 5: `natural.py` (lineas 76–88)

Aquí la cantidad de operaciones es del orden de $len(a) \cdot len(b)$, es decir, `multiplicar(a,b)` tiene complejidad binaria $O(n^2)$, donde $n = len(a) + len(b)$ es el tamaño de la entrada. El peor caso sucede cuando $len(a) = len(b) = n/2$.

Es posible multiplicar números naturales de forma mucho más eficiente. El algoritmo de Karatsuba consiste en reducir un producto de números de n dígitos a sólo 3 productos de números de $d \approx n/2$ dígitos. La clave está en observar que el producto

$$(a_1 10^d + a_0) \cdot (b_1 10^d + b_0) = (c_2 10^{2d} + c_1 10^d + c_0)$$

se puede hacer con sólo 3 multiplicaciones:

$$\begin{aligned}
c_2 &= a_1 \cdot b_1, \\
c_0 &= a_0 \cdot b_0, \\
c_1 &= (a_1 + a_0) \cdot (b_1 + b_0) - c_2 - c_0.
\end{aligned}$$

Cuando los números son suficientemente pequeños (caso base), se multiplican directamente utilizando el algoritmo de la escuela.

La complejidad aritmética $M(n)$ del algoritmo de Karatsuba se puede determinar resolviendo la siguiente recursión:

$$M(n) \leq 3M(\lceil n/2 \rceil) + Cn$$

donde $C > 0$ es una cierta constante que representa las sumas y restas que se necesitan para calcular c_1 y para conseguir el resultado final a partir de c_0, c_1 y c_2.

Para un $n = 2^r$, se puede ver que

$$\begin{aligned}
M(n) = M(2^r) &\leq 3 \cdot M(2^{r-1}) + C \cdot 2^r \\
&\leq 3^2 \cdot M(2^{r-2}) + C \cdot 3 \cdot 2^{r-1} + C \cdot 2^r \\
&\;\;\vdots \\
&\leq 3^r \cdot M(1) + C \cdot (3^{r-1} \cdot 2 + 3^{r-2} \cdot 2^2 + \cdots + 3 \cdot 2^{r-1} + 2^r) \\
&= 3^r \cdot M(1) + C \cdot 3^{r-1} \cdot 2 \cdot \frac{1 - (2/3)^r}{1 - (2/3)} \\
&\leq 3^r \cdot (M(1) + 2 \cdot C) \\
&= O(n^{\log_2 3}).
\end{aligned}$$

Para cualquier n, se puede siempre elegir un $n \leq 2^r < 2n$ y luego aplicar el razonamiento anterior. Se prueba así que $M(n) = O(n^{\log_2 3}) = O(n^{1.585})$ lo que es mucho más eficiente (para n grande) que la multiplicación de la escuela cuya complejidad binaria es $O(n^2)$.

```python
def multiplicar_karatsuba(a, b):
    n = len(a)
    m = len(b)
    if n < m:
        a, b = b, a
        n, m = m, n
    if m <= 10:
        prod = multiplicar(a, b)
    elif m <= n//2:
        c0 = multiplicar_karatsuba(a[:n//2], b)
        c1 = multiplicar_karatsuba(a[n//2:], b)
        c1 = sumar(c1, c0[n//2:])
        if len(c0) < n//2:
            c0 += [0] * (n//2 - len(c0))
        prod = c0[:n//2] + c1
    else:
        c0 = multiplicar_karatsuba(a[:n//2], b[:n//2])
        c2 = multiplicar_karatsuba(a[n//2:], b[n//2:])
        s1 = sumar(a[:n//2], a[n//2:])
        s2 = sumar(b[:n//2], b[n//2:])
        c1 = multiplicar_karatsuba(s1, s2)
        s3 = sumar(c0, c2)
        c1 = restar(c1, s3)
        c1 = sumar(c1, c0[n//2:])
        c2 = sumar(c2, c1[n//2:])
        if len(c0) < n//2:
            c0 += [0] * (n//2 - len(c0))
        if len(c1) < n//2:
            c1 += [0] * (n//2 - len(c1))
        prod = c0[:n//2] + c1[:n//2] + c2
    remover_ceros(prod)
    return prod
```

Programa 6: `natural.py` (lineas 90–121)

Problema 3.1. Implementar la función `div2(a)` que calcule los dígitos de $\lfloor a/2 \rfloor$, es decir, que se comporte como la operación $a//2$ de Python3 pero utilizando la representación de listas de dígitos "reducidas". Utilizar esa función para implementar la función `decimal_a_base2(a)` que devuelve la representación binaria de a (una lista de ceros y unos). Calcular la complejidad binaria de ambas funciones.

Problema 3.2. Implementar la función `base2_a_decimal(b)` que obtenga la lista de dígitos decimales de un número a partir de la lista de sus bits. Calcular la complejidad binaria de esta función.

Problema 3.3. Sea $f : \mathbb{N} \to \mathbb{R}_{>0}$ una función creciente tal que

$$f(n) \leq k \cdot f(\lceil n/r \rceil) + C \cdot n^\alpha \quad \forall n \in \mathbb{N}$$

para ciertos $C, \alpha \geq 0$, $k > 1$ y $r \geq 2$ entero. Demostrar que:

- $\alpha < \log_r k \implies f \in O(n^{\log_r k})$,
- $\alpha = \log_r k \implies f \in O(n^\alpha \log n)$,
- $\alpha > \log_r k \implies f \in O(n^\alpha)$.

Problema 3.4. Demostrar que es posible multiplicar dos matrices de $n \times n$ con $O(n^{\log_2 7})$ operaciones aritméticas. ¿Cuál es la cantidad de operaciones aritméticas si se las multiplica directamente haciendo los productos de filas con columnas?

Problema 3.5. Hacer una gráfica que muestre los verdaderos tiempos de ejecución de la función `multiplicar_karatsuba(a,b)` (medidos en segundos) en función de la cantidad de dígitos totales de entrada $n = len(a) + len(b)$ y comparar con $n^{log_2 3}$.

Problema 3.6. El algoritmo Toom-3 para multiplicar números naturales es una extensión del de Karatsuba dividiendo a cada factor en tres partes. Más precisamente,

$$(a_2 10^{2d} + a_1 10^d + a_0) \cdot (b_2 10^{2d} + b_1 10^d + b_0) = (c_4 10^{4d} + c_3 10^{3d} + c_2 10^{2d} + c_1 10^d + c_0),$$

donde $c_0, \ldots, c_4$ pueden calcularse del siguiente modo:

$$\begin{aligned}
c_0 &= a_0 \cdot b_0,\\
c_4 &= a_2 \cdot b_2,\\
s_1 &= (a_0 + a_1 + a_2) \cdot (b_0 + b_1 + b_2),\\
s_2 &= (a_0 + 2a_1 + 4a_2) \cdot (b_0 + 2b_1 + 4b_2),\\
s_3 &= (a_0 + 3a_1 + 9a_2) \cdot (b_0 + 3b_1 + 9b_2),\\
c_1 &= -\frac{11}{6}c_0 + 3s_1 - \frac{3}{2}s_2 + \frac{1}{3}s_3 - 6c_4,\\
c_2 &= c_0 - \frac{5}{2}s_1 + 2s_2 - \frac{1}{2}s_3 + 11c_4,\\
c_3 &= -\frac{1}{6}c_0 + \frac{1}{2}s_1 - \frac{1}{2}s_2 + \frac{1}{6}s_3 - 6c_4.
\end{aligned}$$

Demostrar que el método es correcto e implementarlo (se necesitará implementar también la función `div6(a)` que equivale a la operación `a//6` de Python3). ¿Cuál es la complejidad binaria de este algoritmo?

Problema 3.7. Reimplementar el algoritmo `gcd_binario(x,y)` del programa `gcds.py` del tema #2 para $x, y \geq 0$ representados por listas de dígitos (reducidas) y demostrar que su complejidad es $O(n^2)$ donde $n = len(x) + len(y)$. Parte del ejercicio consiste en implementar la función `mul2(x)` que calcule el doble de un entero $x \geq 0$.

Problema 3.8. Implementar la función `raiz_cuadrada(x)` que dado un entero (no negativo) calcule su raíz cuadrada $\lfloor \sqrt{x} \rfloor$, utilizando una búsqueda binaria o bien obteniendo dígito por dígito del resultado. Tanto la entrada como la salida del algoritmo serán listas de dígitos. ¿Cuál es la complejidad binaria del método? Generalizar a la función `raiz_k_esima(x,k)` que calcula $\lfloor \sqrt[k]{x} \rfloor$, donde x es una lista de dígitos y $k \geq 1$ un entero.

Hasta ahora hemos visto que la suma, resta y comparación de números naturales de n dígitos puede hacerse en $O(n)$ operaciones elementales entre dígitos. Si se usa el método de la escuela, la multiplicación puede hacerse en $O(n^2)$ operaciones, y si se usa el método de Karatsuba, la cantidad de operaciones se reduce a $O(n^{1.585})$.

El algoritmo de la escuela para dividir enteros largos suele ser presentado informalmente. Para dividir un número a de n dígitos por uno b de $m \leq n$ dígitos, se busca el mayor entero $0 \leq c \leq 9$ tal que $r = a - b \cdot c \cdot 10^{n-m} \geq 0$. Luego se agrega $c \cdot 10^{n-m}$ al cociente y se reemplaza a por r. Ese procedimiento reduce la cantidad de dígitos de a, o en el caso de ser $c = 0$ nos muestra que $a < b \cdot 10^{n-m}$. En este último caso, y suponiendo que $n > m$, se busca el mayor dígito c tal que $r = a - b \cdot c \cdot 10^{n-m-1} \geq 0$ y se añade $c \cdot 10^{n-m-1}$ al cociente. El método termina cuando se llega a $a < b$.

```
123 def division_y_resto(a, b):  # a,b son listas de digitos "reducidas"
124     n = len(a)
125     m = len(b)
126     if n < m:
127         return [], a             # cociente = 0, resto = a
128     q = [0] * (n-m+1)            # crear una lista de ceros para el cociente
129     while comparar(a, b) >= 0:
130         c = 9
131         # la siguiente condicion es equivalente a a<b*c*10^(n-m)
132         while comparar(a[n-m:], multiplicar(b,[c])) < 0:
133             c = c - 1
134         if c != 0:
135             q[n-m] = c
136             a = a[:n-m] + restar(a[n-m:], multiplicar(b,[c]))
137             remover_ceros(a)
138         elif n > m:
139             c = 9
140             # la siguiente condicion es equivalente a a<b*c*10^(n-m-1)
141             while comparar(a[n-m-1:], multiplicar(b,[c])) < 0:
142                 c = c - 1
143             q[n-m-1] = c
144             a = a[:n-m-1] + restar(a[n-m-1:], multiplicar(b,[c]))
145             remover_ceros(a)
146         n = len(a)
147     remover_ceros(q)
148     return q, a
```

Programa 7: `natural.py` (lineas 123–148)

En Python3, la expresión $a[i:j]$ es la lista formada por $a[i], a[i+1], \ldots, a[j-1]$. En caso de no especificarse i, se entiende que $i = 0$, es decir, $a[:j]$ es la lista de las primeras j entradas de a. Del mismo modo, si no se especifica j, se entiende que $j = len(a)$, es decir, $a[i:]$ es la lista de todas las entradas de a a partir de $a[i]$. Esto mismo se puede usar con tuplas.

Es fácil comprobar que `division_y_resto(a,b)` require una cantidad $O(n^2)$ operaciones, donde $n = len(a) + len(b)$ es el tamaño de la entrada. El peor caso sucede cuando $len(b) \approx len(a)/2$.

Parece bastante ineficiente tener que probar con todos los posibles dígitos $c = 9, 8, \ldots, 0$ hasta dar con el correcto en las lineas 8–10 y 15–17. Si bien la cantidad máxima de intentos es 10, eso puede empeorar significativamente si se trabaja con dígitos en una base mas grande.

Suponiendo que el dígito más significativo de b satisface $b_{m-1} \geq 5$, se puede ver que el resultado de las lineas 8–10 será siempre $c = 0$ o $c = 1$ dependiendo de si $a < b \cdot 10^{n-m}$ o no. De forma similar, se puede ver que el resultado de las lineas 15–17 será

$$c^* = \min \left\{ \left\lfloor \frac{a_{n-1} \cdot 10 + a_{n-2}}{b_{m-1}} \right\rfloor, 9 \right\}$$

o $c^* - 1$ o $c^* - 2$. Esto requiere a lo sumo tres intentos hasta acertar con el dígito correcto. Además, este método funciona bien en cualquier base B, suponiendo que $b_{m-1} \geq \lfloor B/2 \rfloor$. A esto se lo llama "división normalizada".

Para conseguir llegar siempre al caso normalizado, se puede premultiplicar a a y b por el dígito $f = \lfloor B/(b_{m-1}+1) \rfloor$ antes de hacer la división. El cociente será el mismo y el resto saldrá multiplicado por f, por lo que al final habrá que dividirlo por f. Por supuesto, para esto habrá que implementar cuidadosamente un algoritmo para dividir por un número de un sólo dígito.

Problema 3.9. Incorporar las mejoras indicadas arriba a la función `division_y_resto(a,b)`. Comparar los tiempos reales de ejecución de ambas versiones.

Es de esperar que el algoritmo de la escuela para hacer divisiones no sea lo más eficiente. A continuación vamos a mostrar que se puede conseguir dividir en $D(n) = O(M(n))$ operaciones, es decir, que la complejidad binaria de la multiplicación y la de la división con resto son del mismo orden. El truco será ver que es posible dividir haciendo unas pocas multiplicaciones.

Digamos que a es de n dígitos y b de m dígitos. Para calcular $\lfloor a/b \rfloor$ podemos primero calcular la expansión decimal de $1/b$ con n dígitos de precisión (es decir, un error absoluto menor que 10^{-n}), luego multiplicar esos n dígitos por a y por último tomar la parte entera. Más precisamente, vamos a calcular un aproximación entera y del número real $10^n/b$ con error absoluto menor que 1 y luego hacer

$$\left\lfloor \frac{a}{b} \right\rfloor = \left\lfloor \frac{a \cdot y}{10^n} \right\rfloor + \left\{ \begin{array}{c} 0 \\ \pm 1 \end{array} \right\}$$

La división entera por 10^n es fácil de hacer, ya que estamos trabajando con la representación en base 10.

Para calcular la aproximación buscada de $1/b$ vamos a utilizar el método de Netwon.

Teorema 3.1. *Sea $f : (r - \varepsilon, r + \varepsilon) \subseteq \mathbb{R} \to \mathbb{R}$ una función analítica en r con radio de convergencia $\varepsilon > 0$, es decir,*

$$f(x) = \sum_{i=0}^{\infty} \frac{f^{(i)}(r)}{i!}(x - r)^i$$

para todo $x \in (r - \varepsilon, r + \varepsilon)$. Supongamos que $f(r) = 0$ y que $f'(r) \neq 0$. Sea

$$\gamma = \sup_{i \geq 2} \left| \frac{f^{(i)}(r)}{i! f'(r)} \right|^{1/(i-1)}.$$

Si $|x - r| < \frac{1}{5\gamma}$, entonces $f'(x) \neq 0$ y $|x - f(x)/f'(x) - r| \leq 5\gamma|x - r|^2 < \frac{1}{5\gamma}$. En particular, si hacemos la iteración $x_{k+1} = x_k - f(x_k)/f'(x_k)$ partiendo de un x_0 tal que $|x_0 - r| < \frac{1}{50\gamma}$, tendremos que $|x_k - r| < \frac{1}{5\gamma}10^{-2^k}$ para todo $k \geq 0$.

Proof. Como f es analítica, f' también lo es (en el mismo intervalo) y su desarrollo en series centrado en r es:

$$f'(x) = \sum_{i=1}^{\infty} \frac{f^{(i)}(r)}{(i-1)!}(x - r)^{i-1}.$$

Suponiendo que $|x - r| < \frac{1}{5\gamma}$, tenemos que

$$\begin{aligned} |f'(x)| &\geq |f'(r)| \left[1 - \sum_{i=2}^{\infty} \left| \frac{f^{(i)}(r)}{(i-1)! f'(r)} \right| \left(\frac{1}{5\gamma} \right)^{i-1} \right] \\ &\geq |f'(r)| \left[1 - \sum_{i=2}^{\infty} i\gamma^{i-1} \left(\frac{1}{5\gamma} \right)^{i-1} \right] = \frac{7}{16}|f'(r)| \end{aligned}$$

y en particular, $f(x) \neq 0$. Además, tenemos que

$$\begin{aligned} \left| x - \frac{f(x)}{f'(x)} - r \right| = \frac{|(x - r)f'(x) - f(x)|}{|f'(x)|} &\leq \frac{16}{7|f'(r)|} \left| \sum_{i=2}^{\infty} \frac{(i-1)f^{(i)}(r)}{i!}(x - r)^i \right| \\ &\leq \frac{16}{7}|x - r|^2 \sum_{i=2}^{\infty}(i - 1) \left| \frac{f^{(i)}(r)}{i! f'(r)} \right| |x - r|^{i-2} \\ &\leq \frac{16}{7}|x - r|^2 \sum_{i=2}^{\infty}(i - 1)\gamma^{i-1} \left(\frac{1}{5\gamma} \right)^{i-2} = \frac{25}{7}\gamma|x - r|^2 \\ &< 5\gamma|x - r|^2 < \frac{1}{5\gamma} \end{aligned}$$

con lo que queda probada la primera parte. Lo de la velocidad de convergencia de la iteración de Newton es hacer inducción utilizando la desigualdad que acabamos de probar. $\square$

En nuestro caso, vamos a utilizar la función $f(x) = \frac{1}{x} - b$ cuyo cero está en el punto $r = 1/b$. Claramente $f(x)$ es analítica en $r = 1/b$ y un cálculo simple muestra que $\gamma = b$. La iteración del método de Netwon es $x_{k+1} = x_k - f(x_k)/f'(x_k) = x_k(2 - bx_k)$ que, afortunadamente, sólo requiere una resta y dos multiplicaciones, pero ninguna división.

Debemos buscar, para empezar, un x_0 tal que $\left|x_0 - \frac{1}{b}\right| < \frac{1}{50b}$. Suponiendo que $10^{m-1} \leq b < 10^m$, es decir, que b tiene m dígitos, resulta que $r = 1/b = 0.0 \cdots 0 * * \cdots$, donde hay exactamente $m - 1$ ceros después del punto decimal. Un x_0 que tenga los primeros tres dígitos no nulos correctos de $1/b$ satisface $\left|x_0 - \frac{1}{b}\right| < 10^{-m-2} < \frac{1}{50b}$ y por lo tanto nos sirve como punto de partida. Si pensamos a $x_0 = y_0 \cdot 10^{-m-2}$ con $y_0 \in \mathbb{N}$ de tres dígitos, lo que necesitamos es que $|y_0 - 10^{m+2}/b| < 1$, es decir,

$$y_0 = \left\lfloor \frac{10^{m+2}}{b} \right\rfloor = \max\left\{t \in \mathbb{N} \;:\; 100 < t \leq 1000,\; bt \leq 10^{m+2}\right\}.$$

El valor de y_0 puede determinarse explorando todos los posibles $101 \leq t \leq 1000$, o bien, haciendo una búsqueda binaria. En cualquier caso, la cantidad de operaciones que se necesitan es $O(m)$.

La iteración de Newton nos garantiza que $|x_k - 1/b| < \frac{1}{5b}10^{-2^k}$, por lo que x_k tendrá al menos 2^k dígitos correctos (comparados con $1/b$) luego de los $m-1$ ceros que vienen a partir del punto decimal. Por otra parte, necesitamos garantizar que $2 + 2^k$ dígitos de x_k estén almacenados correctamente en memoria para que x_{k+1} tenga la precisión indicada. De todo esto, concluímos que x_k será de la forma $y_k 10^{-m-1-2^k}$ con y_k un número natural de $2 + 2^k$ dígitos decimales.

$$\begin{aligned}
y_{k+1}10^{-m-1-2^{k+1}} &\approx y_k 10^{-m-1-2^k}(2 - by_k 10^{-m-1-2^k}) \\
y_{k+1} &\approx y_k 10^{2^k}(2 - by_k 10^{-m-1-2^k}) \\
y_{k+1} &\approx 2y_k 10^{2^k} - by_k^2 10^{-m-1} \\
y_{k+1} &= 2y_k 10^{2^k} - \left\lfloor \frac{\left\lfloor \frac{b}{10^{m-3-2^{k+1}}} \right\rfloor y_k^2}{10^{2^{k+1}+4}} \right\rfloor
\end{aligned}$$

Es fácil ver que la iteración requiere $O(M(2^k))$ operaciones binarias para $k = 0, 1, \ldots, \lfloor \log_2(n) \rfloor$. En la iteración final, obtenemos la aproximación de $1/b$ con n dígitos que buscabamos. La complejidad total del método queda $O(M(n))$.

Veamos un ejemplo concreto. Supongamos que $b = 56273694826793487298234$ de $m = 23$ dígitos decimales.

$$\begin{aligned}
&x_0 = y_0 \cdot 10^{-25} && y_0 = \max\left\{t \in \mathbb{N} \;:\; 101 \leq t \leq 1000,\; bt \leq 10^{25}\right\} = 177 \\
&x_1 = y_1 \cdot 10^{-26} && y_1 = 2 \cdot 177 \cdot 10^1 - \left\lfloor \frac{\lfloor b/10^{18} \rfloor 177^2}{10^6} \right\rfloor = 1778 \\
&x_2 = y_2 \cdot 10^{-28} && y_2 = 2 \cdot 1778 \cdot 10^2 - \left\lfloor \frac{\lfloor b/10^{16} \rfloor 1778^2}{10^8} \right\rfloor = 177703 \\
&x_3 = y_3 \cdot 10^{-32} && y_3 = 2 \cdot 177703 \cdot 10^4 - \left\lfloor \frac{\lfloor b/10^{12} \rfloor 177703^2}{10^{12}} \right\rfloor = 1777029220 \\
&x_4 = y_4 \cdot 10^{-40} && y_4 = 2 \cdot 1777029220 \cdot 10^8 - \left\lfloor \frac{\lfloor b/10^4 \rfloor 1777029220^2}{10^{20}} \right\rfloor = 177702921956329746 \\
&x_5 = y_5 \cdot 10^{-56} && y_5 = 2 \cdot 177702921956329746 \cdot 10^{16} - \left\lfloor \frac{\lfloor b \cdot 10^{12} \rfloor 177702921956329746^2}{10^{36}} \right\rfloor = \\
& && = 1777029219563297453449602028559613 \\
&x_6 = y_6 \cdot 10^{-88} && y_6 = 177702921956329745344960202855961272620268353714009725013100908268
\end{aligned}$$

Sabemos que la diferencia entre $1/b$ y x_6 es menor que 10^{-86}, por lo que si quisieramos dividir un número

$$a = 2934872934729487239488472984749283479283749238427947294923847293847298482014$$

de $n = 76 \leq 86$ cifras por b, bastaría con calcular

$$q = \left\lfloor \frac{a \cdot y_6}{10^{88}} \right\rfloor = 52153549607197851357899473386105573255063608067175345$$

para conseguir el cociente con error de a lo sumo una unidad. Se comprueba que $bq \leq a < b(q+1)$, por lo que no hay que hacer ninguna corrección al valor de q. Por último, el resto se puede calcular haciendo $a - bq = 31237543472563311641284$.

Problema 3.10. Extender la librería `natural.py` con una implementación del algoritmo de la división larga utilizando la iteración de Newton.

Problema 3.11. Demostrar que es posible convertir de base 10 a base 2 y viceversa con $O(M(n))$ operaciones binarias. La idea es calcular primero las potencias 2^{2^i} para $i = 0, 1, \ldots$ en base 10, luego hacer una división con resto por la mayor de esas potencias que sea menor que el número dado, es decir, escribirlo de la forma $a + b2^{2^j}$, y finalmente convertir tanto el cociente b como el resto a a binario.

Problema 3.12. Calcular los primeros mil dígitos de $\sqrt{2}$ después del punto decimal, utilizando la iteración de Newton con la función $f(x) = x^2 - 2$.