Approximating e and fast integer division

1 Approximating e

The constant e is given by the formula

$$e = \sum_{k \geq 0} \frac{1}{k!} \approx 2.71828.$$

It is very well approximated by the nth truncated sum $e_n = \sum_{k=0}^{n} \frac{1}{k!}$ (indeed $e - e_n = \sum_{k>n} \frac{1}{k!} \leq \frac{3}{(n+1)!}$, which ensures that the number of correct digits after comma in e_n is of the order of $n \log(n)$). Note that e_n is a rational number of the form $\dfrac{N_n}{n!}$. The first values are $e_0 = 1,\ e_1 = 2,\ e_2 = \frac{5}{2} = 2.5,\ e_3 = \frac{16}{6} = 2.6667,\ e_4 = \frac{65}{24} \approx 2.7083\ldots$ More generally, we have

$$\frac{N_n}{n!} = e_n = e_{n-1} + \frac{1}{n!} = \frac{N_{n-1}}{(n-1)!} + \frac{1}{n!} = \frac{nN_{n-1}+1}{n!},$$

hence N_n satisfies the recurrence $N_0 = 1$ and $N_n = nN_{n-1} + 1$ for $n \geq 1$.

We have seen in TD2 two methods to compute $n!$ (the denominator of e_n here) : one iterative and one by binary splitting. The second one was faster for large n, where the gain is more significant when using a fast integer product algorithm as the one in the `mpz` library. Similarly we will see here a naive (iterative) and a binary-splitting approach to compute the numerator N_n.

Question 1. Implement a simple iterative function `numer_iter(n)` that computes N_n. To test your method, check for instance that for $n = 20$ it outputs 6613313319248080001, and run the tests in `test.py`.

Recall that the binary-splitting approach to compute $n!$ was to define $P(a,b) = \prod_{i=a}^{b} i$ and use the identities

$$P(a,b) = \begin{cases} a & \text{for } a = b, \\ P(a,m)P(m+1,b) & \text{for } a < b, \text{ where } m = \lfloor (a+b)/2 \rfloor \end{cases}$$

and moreover $n! = P(1,n)$. We have included the solution to the function `factor_bin(n)` in the file `exponential.py` (note that this implementation is done so as to perform multiplications using the `mpz` library).

Something similar can be done to compute N_n, but as we will see the multiplications are to be performed on 2×2 matrices. Note that $e_n - e_{n-1} = \frac{1}{n!}$, hence for $n \geq 2$ we have $e_n - e_{n-1} = \frac{1}{n}(e_{n-1} - e_{n-2})$, which gives $e_n = \frac{1}{n}((n+1)e_{n-1} - e_{n-2})$. Hence if we define the matrix A_n and the vector V_n as

$$A_n = \begin{bmatrix} n+1 & -1 \\ n & 0 \end{bmatrix}, \quad V_n = \begin{bmatrix} e_n \\ e_{n-1} \end{bmatrix},$$

then we have $V_n = \frac{1}{n} A_n V_{n-1}$ for $n \geq 2$. Hence for $n \geq 2$, we have $V_n = \frac{1}{n!} A_n A_{n-1} \cdots A_2 V_1$, or equivalently

$$n! V_n = M_n V_1, \qquad \text{with } M_n := A_n A_{n-1} \cdots A_2.$$

Since $n! V_n = \begin{bmatrix} n!\, e_n \\ n!\, e_{n-1} \end{bmatrix} = \begin{bmatrix} N_n \\ nN_{n-1} \end{bmatrix}$, we conclude that N_n is the first component of the vector $M_n V_1$ (with $V_1 = \begin{bmatrix} e_1 \\ e_0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$).

Question 2. Based on this, implement a binary-splitting function `numer_bin(n)` that computes N_n, after computing the 2×2 matrix M_n (be careful that the matrix product is not commutative!). You should store a 2×2 matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ in Python as $[[a, b], [c, d]]$, and write your own function `prod_mat(K,L)` for the multiplication of two 2×2 matrices (do not use the `np` data structures to store matrices). Another important point is that your algorithm for N_n should use integer multiplications using the `mpz` library (see the code of `facto_bin(n)`, note that the `mpz` syntax is used just when $ns = ne$ in the auxiliary recursive function). Tests your code using `test.py`.

Once your function works, you can compare running times by using the (already provided) function `compare_times_numer()`. The binary-splitting function should become better only for very large sizes (typically $n \geq 65,536$).

Question 3. It is quite easy to see that $e - e_n \leq \frac{3}{(n+1)!}$, hence $e - e_n < 10^{-k}$ for any k such that $k \log(10) > \log((n+1)!) - \log(3)$, i.e., such that $k \log(10) > \sum_{i=1}^{n+1} \log(i) - \log(3)$. Hence, if we let $k \geq 1$, then we have $e - e_n < 10^{-k}$ for *the smallest* n such that

$$\sum_{i=1}^{n+1} \log(i) > k \log(10) + \log(3).$$

Write a function `exp_digits(k)` that acurately computes e up to k digits. In other words it has to output the integer $\lfloor e * 10^k \rfloor$. (Note : for two `mpz` integers a, b, `a//b` gives the `mpz` integer for $\lfloor a/b \rfloor$). Test your function using `test.py`. Also a list of 10,000 correct digits of e can be found at `http://www-history.mcs.st-and.ac.uk/HistTopics/e_10000.html`.

2 Euclidean division using Newton's iterations

We would like to write our own division algorithm `own_div(A,B)` to compute $\lfloor A/B \rfloor$ (usually done by the command `A//B` in Python), using Newton's iterations. We are here only allowed to use integer additions and multiplications, and bit-shift operators (e.g. `x>>k` gives `x//(2**k)` and `x<<k` gives `x*(2**k)`). To simplify we will write a first version of the algorithms without `mpz`. (Once it works it is then easy to do the few code modifications to indicate that integers are to be treated by `mpz`).

In a first step you will have to implement a DAC algorithm to compute the so-called *dual* of an integer. For x a positive integer, the *size* n of x is defined here as the bit-length of x (available as `x.bit_length()` in Python), i.e. n is the unique positive integer such that $2^{n-1} \leq x < 2^n$. The *dual* of x is the integer

$$d(x) := \lfloor 2^{2n-1}/x \rfloor.$$

Note that $2^{n-1} < \frac{2^{2n-1}}{x} \leq 2^n$, hence $d(x)$ has also size n (except in the special case $x = 2^{n-1}$, where $d(x) = 2^n$ has size $n+1$). The function `dual_for_tests(x)` (to be used only for tests!) computes $d(x)$. For instance $d(1) = 2$, and $d(7) = \lfloor 2^5/7 \rfloor = 4$.

For $n \geq 2$, we let $n_2 := \lceil n/2 \rceil$, and let x_2 be the integer (of size n_2) made of the n_2 leftmost bits of x, i.e. $x_2 = \lfloor x/2^{\lfloor n/2 \rfloor} \rfloor$. Let $a = \frac{x}{2^{n-1}}$ (so $1 \leq a < 2$). By definition of $d(x)$ we have

$$0 \leq \frac{2^{2n-1}}{x} - d(x) < 1, \text{ hence } 0 \leq \frac{1}{a} - d(x)2^{-n} < 2^{-n}.$$

Hence, $d(x)$ can be seen as giving the n most significant bits (after comma) of $1/a$. Similarly, $d(x_2)$ gives the n_2 most significant bits of $1/a$. Hence, from the quadratic convergence of the Newton iteration to compute $1/a$ (see slide 7 in this week's lecture, and slide 12 for the analogous to power series), we expect that if we let $y' = d(x)2^{-n}$ and let $y = d(x_2)2^{-n_2}$ then we should have

$$y' \approx \mathcal{N}(y), \quad \text{where } \mathcal{N}(y) := y + y(1 - ay) = 2y(1 - \tfrac{a}{2}y).$$

In other words we should have (recall that $\frac{a}{2} = \frac{x}{2^n}$)

$$d(x) \approx \left[2d(x_2) \cdot (2^{n+n_2} - x \cdot d(x_2))\right] // 2^{2n_2}.$$

If we let $\tilde{d}(x)$ be the right-hand side, we do not always have $d(x) = \tilde{d}(x)$, but we expect to be close. In case $r(x) := 2^{2n-1} - \tilde{d}(x)x$ does not lie in the desired interval $[0, \ldots, x-1]$, we can just add (if $r(x) < 0$) or subtract (if $r(x) > x$) to $r(x)$ enough copies of x to reach $[0, \ldots, x-1]$, the number of copies we have used indicates the distance between $\tilde{d}(x)$ and the correct value $d(x)$.

Question 4. Based on this, implement the function `dual_dac(x)` that computes $d(x)$ using a DAC approach. Specifically, your function should recursively compute $d(x_2)$, then compute $\tilde{d}(x)$ and correct it so that $2^{2n-1} - \tilde{d}(x)x$ lies in $[0, \ldots, x-1]$; return the corrected value. To test it, you can check that it gives the same results as `dual_for_tests(x)`, and also run tests in `test.py`. It is also interesting to print the intermediate value $\tilde{d}(x)$. You should see that it is often already exactly at $d(x)$, or otherwise very close to it.

We can now complete our algorithm to compute $\lfloor A/B \rfloor$. The trick is to multiply A by the dual of $2^k B$, for some well-adjusted k. Precisely, let p be the size of A and q the size of B. For $k \geq 0$, we let C_k be the dual of $2^k B$. By definition we have

$$0 \leq \frac{2^{2q+2k-1}}{2^k B} - C_k < 1, \quad \text{hence } 0 \leq \frac{1}{B} - \frac{C_k}{2^{2q+k-1}} < \frac{1}{2^{2q+k-1}}, \quad \text{hence } 0 \leq \frac{A}{B} - \frac{AC_k}{2^{2q+k-1}} < \frac{A}{2^{2q+k-1}}.$$

Hence, for any $k \geq 0$ such that $2q + k - 1 \geq p$, we have $0 \leq \frac{A}{B} - \frac{AC_k}{2^{2q+k-1}} < 1$, which implies that $\lfloor A/B \rfloor - \lfloor AC_k/2^{2q+k-1} \rfloor$ is either 0 or 1. Let $Q := \lfloor AC_k/2^{2q+k-1} \rfloor$, then the previous inequality shows that the correct value for $\lfloor A/B \rfloor$ is Q whenever $A < B(Q+1)$ and $Q+1$ otherwise.

Question 5. Based on this, implement the function `division(A,B)` that computes $\lfloor A/B \rfloor$, and test that it gives the same result as `A//B`. Test your function using `test.py`.