Remarks You can use whatever programming language you want to solve the exercises. However, these notes are focused on Python, so you may need to adapt them to fit your language of choice. If you do not know some notions or algorithms that we mention here, do not hesitate to search on the Internet for information.

Euclid and Bézout

Exercise 1.

(a) Write a function `gcd(m,n)` that computes the GCD of two natural numbers using the exhaustive search (i.e., by testing all the numbers).

(b) Test the function `gcd(m,n)` on different numbers to make sure that it outputs correct results.

(c) Write a function `gcd_euclid(m,n)` that computes GCD using the Euclid algorithm.

(d) Test the function `gcd_euclid(m,n)`.

(e) Compare the execution time of the functions `gcd(m,n)`, `gcd_euclid(m,n)`, and `math.gcd(m,n)` (a Python function computing GCD). To do so, for every number m between 1 and 1000 pick a random number n that is between $\sqrt{m}$ and $m-1$. Make a graph of the curves showing the execution time for all three methods. What do you observe ? (Hint : you may use the function `random.randint` to pick random numbers, `time.time` to measure time, `matplotlib.pyplot` to make the graph, and `math.sqrt` to compute the square roots.)

(f) Alice wants to know what is the probability that two random integers are coprime (i.e., have GCD equal to 1). In order to help her, pick random integers from the interval $[1, 10^8]$ and compute their GCD. Do this 10^6 times and let $\hat{\rho}$ denote the proportion of coprime pairs. Compute the number $\sqrt{6/\hat{\rho}}$. How do you think—what is the answer to Alice's question ?

(g) Python offers a library called `numpy.polynomial` that handles operations on polynomials. Read about the basic operations that this library offers (addition, multiplication etc.). How would you implement a function that computes the GCD of two polynomials ? Test it on some examples.

From now on, we will not ask you explicitly to test the functions that you write : you should *always* do that. Test your function *just after* you wrote it (do not wait to have 10 functions before you start testing) and do not hesitate to do *many* tests.

Exercise 2. We now want to write a procedure that takes as an input two natural numbers m, n and finds two integer numbers p, q such that $pm+qn = \gcd(m, n)$. The existence of such pair is given by the Bézout identity. It is important to note that one of the numbers p, q may be negative. For instance, if $(m, n) := (2, 3)$, then we have $(p, q) = (1, -1)$. As previously, we would like to start by the exhaustive search that enumerates all possible numbers. This is more difficult than before, since we need to enumerate pairs of signed numbers.

(a) There is a natural way of enumerating all integer numbers : $0, 1, -1, 2, -2, 3, -3, \ldots$ Write a function `all_integers(k)` that takes a natural number k as an input and outputs the kth element of the sequence given above (note that the index of the first element of the sequence is 0).

(b) Write a function `natural_pairs(k)` that enumerates all pairs of natural numbers. A reasonable way of enumerating such pairs is given in Figure 1 and was studied by Cantor. He noted that the coordinates (x, y) of the kth element of this sequence can be computed using the formulas $w := \lfloor(\sqrt{8k+1} - 1)/2\rfloor$, $t := (w^2 + w)/2$, $y := k - t$, $x := w - y$. You can use these formulas to implement your function. (If you want to know more on this topic, search for "pairing functions" on the Internet.)

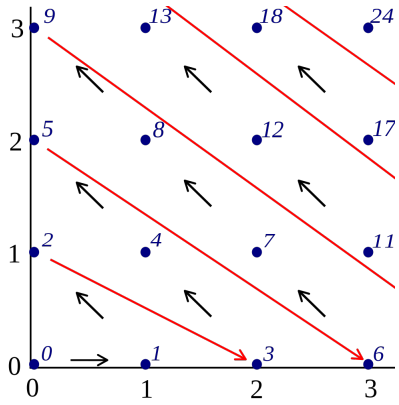

FIGURE 1 – The Cantor pairing function. Source : Wikipedia

(c) Write a function `integer_pairs(k)` that enumerates all pairs of integer numbers. To do so, take the output of `natural_pairs(k)` and use `all_integers()` on both coordinates.

(d) Write a function `exhaustive_bezout(m,n)` that computes the numbers (p, q) from the Bézout identity using exhaustive search.

(e) Write a function `bezout(m,n)` that does the same thing but without using exhaustive search. What is the complexity of your function ? (Hint : proceed similarly to the Euclid algorithm.)

(f) Compare the execution time of the two methods.

Modular exponentiation and prime numbers

Exercise 3.

(a) Write a function `power(a,b)` that takes two natural numbers a, b as an input and outputs a^b using a naive algorithm (a loop that does b multiplications).

(b) Write a function `fast_power(a,b)` that computes a^b using exponentiation by squaring.

(c) Write a function `power_modulo(a,b,c)` that takes numbers a, b, c as an input and computes $(a^b \mod c)$. Consider two options : compute a^b and then take the reminder modulo c or use modular arithmetic for all computations, not only at the end. Which of these methods is better ? Compare your functions with the `pow` function available in Python.

Exercise 4.

(a) Write a function `is_prime(n)` that checks if a given number is prime. What is the complexity of your function ? Use of the Landau notations : $O(\cdot)$, $\Theta(\cdot)$, $\Omega(\cdot)$, etc.

(b) Improve the function `is_prime(n)` so that its complexity is proportional to $\sqrt{n}$.

(c) Write a function `erasthotenes(n)` that outputs a list of all primes smaller than n. To do so, use the sieve of Erasthotenes.

(d) The Fermat little theorem states that if p is a prime number, then

$$a^p \equiv a \mod p \tag{1}$$

for every natural number a. However, prime numbers are not the only ones that have this property. The numbers that are not prime but satisfy it are called Carmichael numbers. Write a function `charmichael_test(p,N)` that tests if a given number p is prime and, if not, checks if it satisfies the equality (1) for all natural numbers $1 \leq a \leq N$. Take $N = 100$ and try to find all Carmichael numbers smaller than 2000. Test your candidates by putting $N = 10000$. What do your tests indicate ?