University of Twente
Module 7: Discrete Structures and Efficient Algorithms, 2016

Assignments Intro Python (February 2, 2016)

It is a good habit to create (at least) one Python file per exercise.

Exercise 1: *Multiples of 3 and 5* (from Project Euler 1)
If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

→ Find the sum of all the multiples of 3 or 5 below 1000.

Exercise 2: *Euclidian algorithm*
The Euclidian algorithm can be used to find the *greatest common divisor* (GCD) of two numbers, in an efficient way.
The algorithm is as follows (in pseudo-code):

```
function gcd(a, b)
    while a =/= b
        if a > b
           a := a - b;
        else
           b := b - a;
    return a;
```

→ Implement the Euclidian algorithm in Python.

To check your algorithm: the GCD of 3141 and 156 is 3.

→ If you where to have a rectangle with size $12345678 \times 987654321$, what area would the largest square tile have with which you can cover the entire rectangle without leaving gaps?

→ Implement a function `frac` which takes the arguments `a` and `b` and calculates the smallest form of the fraction a/b.

Make sure `b` cannot be 0. Make sure your function can handle negative fractions properly.

→ *Optional:* implement the Extended Euclidian algoritm, which calculates the integers x and y (called Bézout coefficients) in the equation $ax + by = gcd(a, b)$ (given a and b):

```
function extended_gcd(a, b)
    s := 0;    old_s := 1
    t := 1;    old_t := 0
    r := b;    old_r := a
    while r =/= 0
        quotient := old_r div r
        (old_r, r) := (r, old_r - quotient * r)
        (old_s, s) := (s, old_s - quotient * s)
        (old_t, t) := (t, old_t - quotient * t)
    output "Bezout␣coefficients:", (old_s, old_t)
    output "greatest␣common␣divisor:", old_r
    output "quotients␣by␣the␣gcd:", (t, s)
```

where `(old_r, r) := (r, old_r - quotient * r)` is equal to

```
temp := r;
r := old_r - quotient * temp;
old_r := temp;
```

and `div` means dividing an integer without considering the remainder (integer division).

Exercise 3: *Different number system*
There are a few aliens living on the Planet Mars. Humans from Earth (NASA) have set up communications with the aliens, using only numbers. However the communications are not going well: the Martians use another number system. You must help NASA communicating with the aliens.

→ Write a function `encode` that converts a number between 0 and 35 to a number or (capital) letter.

  For example, input 0 must return `0`, input 8 must return `8` and input 35 must return `Z`.

→ Write a function `toK` that takes a positive number n (in the decimal system) and an integer $2 \leq k \leq 35$ and converts it to a string in a k-digit number system.

  For example: the input $n = 100, k = 10$ must return `100` and the input $n = 4321, k = 16$ must return `10E1`.

→ Write a function `decode` that converts a string containing one number (`0-9`) or letter (`A-Z`) to a number between 0 and 35. The function must be the inverse of `encode`.

→ Write a function `fromK` with two arguments s and k that converts a string of letters and numbers s from a k-digit number system to the decimal number system. The function must be the inverse of `toK`.

→ Use the functions `toK` and `fromK` to create a function `convert` with arguments, k, m and s, which converts the string s from a k-digit decimal system to an m-digit decimal system.

  For example: the input $k = 2, m = 4$ and $s =$ `10011010` outputs `2122`, and the input $k = 16, m = 7$ and $s =$ `B48C03` is `202400366`.

Let's hope you have solved NASA's problem, and we can now successfully communicate with the Martians!
For more information on bases and numeral systems, see e.g. `http://en.wikipedia.org/wiki/Radix`.

Exercise 4: *Fibonacci* (from Project Euler 2)
Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

$$1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

→ By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

Exercise 5: *Palindromes* (from Project Euler 4)
A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is $9009 = 91 \times 99$.

→ Find the largest palindrome made from the product of two 3-digit numbers.

Exercise 6: *Factorials!*
The factorial of n is defined as

$$n! = \prod_{k=1}^{n} k,$$

with $0! = 1$.

→ Write a function `fact` that takes an integer argument $n \geq 1$ and outputs $n!$.

For example: $28! = 304888344611713860501504000000$.

The binomial $\binom{n}{k}$ is defined as

$$\frac{n!}{(n-k)!\,k!}.$$

→ Write a function `binom` that takes two integer arguments $n \geq 1$ and $n \geq k \geq 1$ and outputs $\binom{n}{k}$.

For example: $\binom{12}{8} = 495$ and $\binom{40}{2} = 780$. It is possible to think of smarter ways to implement the binomial function than simply calculating the factorials.

Exercise 7: *Prime tester and generator*
A prime is an integral number of which has no divisors other than 1 and itself. The number 1 is not a prime. The first few primes are

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, \ldots$$

For many purposes it is useful to be able to test if a given number is prime. For small numbers this is no problem, but for

$$5487513338684751927310969315420497039547508092093535558024525292334330593900490\!3$$

this is harder, even with a computer (it is a prime though). For numbers larger than 10^{20} there is no efficient enough method to quickly test if a number is prime. However, there are *probabilistic* prime tests: the algorithm outputs either *no*, or *maybe*. For a defined number of runs, the probability that the *maybe* means *yes* or the *maybe* means *no* is known.
One of these tests is the Miller Rabin test. It uses the interesting property that

$$a^{(n-1)} \equiv 1 \mod n$$

for a prime number n and a number a which n does not divide. The algorithm goes as follows:

```
Input: n > 3, an odd integer to be tested for primality;
Input: k, a parameter that determines the accuracy of the test
Output: composite if n is composite, otherwise probably prime

write n - 1 as d 2^r with d odd by factoring powers of 2 from n - 1
WitnessLoop: repeat k times:
   pick a random integer a in the range [2, n - 2]
   x <- a^d mod n
   if x = 1 or x = n - 1 then do next WitnessLoop
   repeat r - 1 times:
      x <- x^2 mod n
      if x = 1 then return composite
      if x = n - 1 then do next WitnessLoop
   return composite
return probably prime
```

To get random numbers you can use the `random` library. Write `import random as rand` at the top of your Python file to include the library. Then you can use the function

$$\texttt{rand.random()} \sim U(0, 1)$$

which generates numbers in the range $[0, 1)$, distributed uniformly.
Make sure you use the `pow(a, e, n)` function defined in Python to calculate

$$a^e \mod n$$

because it is much faster than writing `a**e % n`.

→ Implement the Miller-Rabin test, by writing a function `isPrime` that takes an integral odd input $n \geq 3$ and outputs whether it is prime. Test it using $k = 10$ on some (very large) odd numbers $n > 3$, and use WolframAlpha to check whether they are prime.

  Some very large primes to help testing:
  669483106578092405936560831017556154622901950048903016651289
  7595009151080016652449223792726748985452052945413160073645842090827711
  18532395500947174450709383384936679868383424444311405679463280782405796233163977

→ Write a function that starts at a given number m and finds the first integer $n \geq n$ that is probably prime.

Exercise 8: *(Optional) Prime sieve*
While checking if a given number is useful, and one can generate a few (large) prime numbers with it, other applications require all primes between 2 and a limit N. A quick method is the *Prime Sieve (of Eratosthenes)*, which *sieves* the non-prime (composite) numbers from the range $2..N$. The Greek Eratosthenes found this method 240BC, and it is still the base for the most efficient prime sieves up-to-date.
The algorithm is as follows:

```
Create a list of booleans isComposite to store n-1 booleans: 2 .. N,
    initially all false
Create an empty list of primes
For all numbers m: 2 .. N:
     if isComposite[m] is false:
          add m to the list of primes
          set all isComposite[k] to true, for k * m <= N, k >= 2
     otherwise:
          it is a composite number
```

→ Write a function that generates the primes up to a limit N, using the Prime Sieve of Eratosthenes.

Exercise 9: *(Optional) Matrices and lists*

→ Initialize a 3×4 matrix A as follows:

```
A=[[0,1,2,3],[4,5,6,7],[8,9,10,11]]
```

→ Change elements of the matrix using `A[i][j]` ($0 \leq i < 3, 0 \leq j < 4$). Use `print(A)` to print A.

→ Do the same with two lists v and w with zero's on all 30 positions:

```
v=[0]*30
w=[0]*30
```

This notation is called short list notation. Change a few elements of the list, and compute the sum of all elements of both lists.

→ We will now do the same for a matrix. Create a 10×10 matrix A using:

```
A=[[0]*10]*10
```

Set one or more values to a position in the matrix. Print the matrix. What went wrong?

Solve this problem: write a few lines of code that can initialize an $n \times n$ matrix with zeros, for given integers n and m. (Using a `for`-loop seems unavoidable, although short list notation is also possible.)

Exercise 10: *(Optional) Prime factor* (from Project Euler 3)
The prime factors of 13195 are 5, 7, 13 and 29.

→ What is the largest prime factor of the number 600851475143?

Exercise 11: *(Optional) Divisors* (from Project Euler 5)
2520 is the smallest number that can be divided by each of the numbers from 1 to 10 without any remainder.

→ What is the smallest positive number that is evenly divisible by all of the numbers from 1 to 20?

This problem can be solved using either pen and paper (pure mathematics), or a computer (mathematics and programming). Try both ways!