Introduction to Programming Homework 4

Due Thursday Oct 20 by 12h00

You will turn in your homework via e-mail (andrew.yarmola@uni.lu). For this homework, you will work in a text editor of your choosing. See instructions from the previous homework on how to write modules. **All of your functions in this homework MUST have DOCSTINGS.**

Exercise 1 (Polynomials)

Write a module called `poly.py`. We will think of a degree k polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k$$

with **float** coefficients as a tuple (or list) of **floats** `(a_0, a_1, a_2, ..., a_k)` of length `k+1`. Given polynomials $p(x)$ and $q(x) \neq 0$, we can always find polynomials $t(x), r(x)$ with real coefficients such that

$$p(x) = t(x)\,q(x) + r(x) \qquad \text{and} \qquad \deg(r) < \deg(q).$$

In this exercise we will just use floating-point arithmetic (i.e. the standard python `+,-,\*,/`), so the above will be *almost* an equality.

- **a.** Implement a `remainder(p,q)` function which returns the remainder polynomial `r` when dividing `p` by `q`. Raise an exception on bad input. Here is an example,

In [1]:

```
import poly
poly.remainder((0.,1.,2.,3.,4.), (1.,1.,2.))
```

Out[1]:

```
(0.25, 0.75)
```

- **b.** Implement a recursive function `gcd(p,q)` to return the monic greatest common divisor of `p,q`. Here is an example,

In [2]:

```
poly.gcd((1.,1.,2.,3.,1.), (0.,1.,1.))
```

Out[2]:

```
(1.0, 1.0)
```

Note: We will implement finite fields and polynomials over finite fields in a later homework.

Exercise 2 (Tower of Hanoi)

The Tower of Hanoi is an interesting mathematical puzzle. We start with three pegs and a number of disks of different (distinct) sizes which we can slide onto any peg. The puzzle starts with the disks in a stack on one peg in ascending order of size with the smallest at the top.

The objective of the puzzle is to move the entire stack to another peg while obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack
- No disk may be placed on top of a smaller disk.

Create a module called `hanoi.py`.

- **a.** Write a **recursive** function `num_steps(n)` to compute the number of steps it takes to solve the puzzle with `n` disks. Here is an example,

In [3]:

```
import hanoi
hanoi.num_steps(5)
```

Out[3]:

```
31
```

- b. Label the pegs `'a','b','c'` and label the disks with numbers `1, ..., n` from smallest to largest. A **state** will be a dictionary `{'a' : list_of_disks_on_a, 'b': list_of_disks_on_b , 'c' : list_of_disks_on_c}` with each list ordered from smallest disk to largest. For example, the puzzle starts with `{'a' : [1,2,...,n], 'b' : [], 'c' :[]}`. Write a **recursive** function `solution_states(n)` which **returns a list** of states solving the problem (beginning with the start state). For example,

In [4]:

```
steps = hanoi.solution_states(3)
print(*steps, sep='\n') # we will talk about the *-notation
```

```
{'b': [], 'a': [1, 2, 3], 'c': []}
{'b': [], 'a': [2, 3], 'c': [1]}
{'b': [2], 'a': [3], 'c': [1]}
{'b': [1, 2], 'a': [3], 'c': []}
{'b': [1, 2], 'a': [], 'c': [3]}
{'b': [2], 'a': [1], 'c': [3]}
{'b': [], 'a': [1], 'c': [2, 3]}
{'b': [], 'a': [], 'c': [1, 2, 3]}
```

- **Hint:** You will (most likely) need to use `copy.deepcopy` from the module `copy`.

Exercise 3 (Collatz continued)

Create a module `collatz.py`. In a previous homework you wrote a function to compute the length of the Collatz sequence for a given number `n`. Now, we will try to find the **longest** Collatz sequence length.

- **a.** Write a function `collatz_max(n)` which takes a number `n` and returns a pair `(a, s)`, where `s` is the length of the longest Collatz sequence among all integers `x` with `0 < x < n` and `a` is the starting point of such a sequence. For example `collatz_max(25) = (18, 20)`

- **b.** What is `collatz_max(10000000)`? **Hint :** make use of the tools from Lecture 4 so that your code only runs for a few seconds.

Exercise 4 (Randomized Fermat Primality Test)

Create a module `prime_tests.py`. You will implement the randomized Fermat primality test. Recall

Theorem (Fermat's little theorem). Let $n > 1$ be an integer, then n is prime if and only if for all integers $0 < x < n$ one has

$$x^{n-1} \equiv 1 \bmod n$$

- **a.** Write a function `satisfies_fermat(n)` which takes a number `n` and returns `True` if `n` satisfies the conclusion of Fermat's little theorem, and `False` otherwise. Use the `assert` statement to make sure `n` is greater than zero. If you test all the numbers $0 < x < n$ in random order, this should speed up your elimination.

A number x with $0 < x < n$ is called a **Fermat witness for** n if $x^{n-1} \not\equiv 1 \bmod n$. To show that a number n is composite, we just need to find one Fermat witness. A natural question to ask is what portion of the numbers $\{1, 2, \ldots, n-1\}$ are Fermat witnesses of n.

Lemma. If a composite number n has at least one Fermat witness x such that $\gcd(x, n) = 1$, then at least half of the elements of $\{1, 2, \ldots n-1\}$ are Fermat witnesses for n.

Thus, if n has at least one **relatively prime** Fermat witness, then we have a 50% chance of concluding n is composite by randomly choosing $x \in \{1, 2, \ldots n-1\}$.

Definition. An odd composite number n is a **Carmichael number** if every x satisfying $\gcd(x, n) = 1$ has $x^{n-1} \equiv 1 \bmod n$. A number is called **Fermat pseudoprime** if it is either prime or a Carmichael number.

Using these results, we can construct a probabilistic primality test as follows. Given odd n, uniformly pick k distinct number $x_i \in \{1, 2, \ldots n-1\}$ with $i = 1, \ldots, k$. If $x_i^{n-1} \equiv 1 \bmod n$ for all $i = 1, \ldots, k$, then n has probability (at least) $1 - 1/2^k$ of being a Fermat pseudoprime. **Note :** actually, the probablity is much better because we choose x_i to be distinct.

- **b.** Write a function `probably_fermat_pseduoprime(n, prob)` which takes a number `n` and a probability `prob` and returns `True` if `n` has probability `prob` (or more) of being a Fermat pseudoprime. In your test, make sure you **choose your `x`'s randomly** (using the `random` module). Use the `assert` statement to make sure `n` is greater than zero.