Introduction to Programming Homework 4 Solutions

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.

In [19]:

```
def is_num(x) :
    """ Returns True if x is an int, float, or complex. """
    return isinstance(x, (int, float, complex))

def strip_tail_zeros(p) :
    """ Removes zeros from the end of a list p. Returns
        the number of zeros stripped. """
    count = 0
    while len(p) > 0 and p[-1] == 0. :
        p.pop()
        count += 1
    return count

def deg(p) :
    """ Retruns the degree of a polynomial p,
        prepresented as a sequence of floats. """
    return len(p) - 1

def scale(p, a, divide = False) :
    """ By default, multiplies every element of list p by a.
        If divide = True, then divides every element
        of p by a. p is assumed to be a list of floats. """
    for i,v in enumerate(p) :
        if divide :
            p[i] = v/a
        else :
            p[i] = a*v

def subtract_with_shift(p, q, shift) :
    """ Treating p, q as polynomials p(x), q(x)
```

```
        Treating p, q as polynomials p(x), q(x)
        this method returns the polynomial
        p(x) - q(x)*x^shift. """
    if type(shift) is not int or shift < 0 :
        raise ValueError("Shift must be a positive int")
    len_diff = len(q) + shift - len(p)
    if len_diff > 0 :
        p.extend( [0.0] * len_diff )
    for i, v in enumerate(q) :
        p[i+shift] -= v

def remainder(p,q) :
    """ Treating the sequences of floats p,q as
        polynomials, we return the remainer of
        dividing p by q using floating-point
        arithmetic. """
    if False in map(is_num, q) or \
       False in map(is_num, p) :
        raise ValueError("Input is not polynomial")
    r = list(p)
    q_monic = list(q)
    # strip starting zeros
    strip_tail_zeros(r)
    strip_tail_zeros(q_monic)
    if len(q_monic) == 0 :
        raise RuntimeError("Division by zero polynomial")
    # make q_monic actually monic
    scale(q_monic, q_monic[-1], divide = True)
    while deg(r) >= deg(q_monic) :
        t = list(q_monic)
        scale(t, r[-1])
        subtract_with_shift(r, t, deg(r) - deg(t))
        if strip_tail_zeros(r) == 0 :
            raise RuntimeError("Floating point arithmetic failed")

    return tuple(r)
```

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

In [20]:

```
def gcd(p,q) :
    """ Recursive implementation of the gcd
        algorithm for polynomials p,q.
        Returns the monic gcd. """
    if deg(q) > deg(p) :
        p, q = q, p # this is a stanfrad way to swap things
    r = remainder(p,q)
    if len(r) == 0 :
        t = list(q)
        scale(t, t[-1], divide = True)
        return tuple(t)
    else :
        return gcd(q,r)

def gcd_v2(p,q) :
    """ Non-recursive implementation of the
        gcd algorithm for polynomials p,q.
        Returns the monic gcd. """
    if deg(q) > deg(p) :
        p, q = q, p # this is a stanfrad way to swap things
    prev = tuple(q) # a copy
    curr = remainder(p,q)
    if len(curr) > 0 :
        prev, curr = curr, remainder(prev, curr)

    t = list(prev)
    scale(t, t[-1], divide = True)
    return tuple(t)
```

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.

In [11]:

```
def num_steps(n) :
    """ Returns the number of steps needed to solve
    the Tower of Hanoi puzzle with n disks. """
    assert n > 0 and type(n) is int
    if n == 1 :
        return 1
    else :
        return 2 * num_steps(n-1) + 1
```

- 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).

In [12]:

```
import copy

def append_to_peg(state, peg, value) :
    """ Appends value to the end of the list state[peg].
    Does *not* check if value > max(state[peg])! """
    state[peg].append(value)
    # we don't return anything because we modified state

def switch_pegs(state, some_peg, other_peg) :
    """ Switches values of some_peg and other_peg in state. """
    switched = { some_peg : state[other_peg],
                 other_peg : state[some_peg] }
    state.update(switched)
    # we don't return anything because we modified state

def solution_states(n) :
    """ Prints a list of solution states (which are dictionaries)
        to solve the Tower of Hanoi problem with n disks """
    assert n > 0 and type(n) is int
    if n == 1 :
        sol_steps = [ { 'a' : [1], 'b' : [ ], 'c' : [ ] } ,
                      { 'a' : [ ], 'b' : [ ], 'c' : [1] } ]
        return sol_steps
    else :
        # we will use the solution from n - 1 to first
        # move the top n - 1 to peg 'b', then move disk n
        # to peg 'c', and finally reuse the n - 1 solution
        # to move everything to peg 'c'
        start = solution_states(n-1)
        # we make a deep copy so that when we modify start's
        # elements, we don't change what's in finish !
        finish = copy.deepcopy(start)

        for i in range(len(start)) :
            # we add disk n to the 'a' peg and switch 'b', 'c' pegs
            # so that we are moving disks from 'a' to 'b'
            switch_pegs(start[i], 'b', 'c')
            append_to_peg(start[i], 'a', n)
            # we must move disk n to peg 'c' and then use the n - 1
            # solution to move everything from peg 'b' to 'c'.
            # we accomplish this just making finish move disks from 'b'
            # to 'c' and making sure to add disk n onto peg 'c'
            switch_pegs(finish[i], 'a', 'b')
            append_to_peg(finish[i], 'c', n)

        return start + finish
```

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)`

In [13]:

```
def collatz_max(n) :
    """ Returns a pair (a,s) where s in the maximum length
        of a Collatz sequence starting with 0 < x < n and a is
        a starting point of such a sequence. """
    # we use a dictionary to keep track of all of out known
    # Collatz sequence lengths
    assert n > 0 and type(n) is int
    known = {1 : 0}
    max_steps = 0
    max_start = 1
    for m in range(1, n) :
        m_steps = 0
        # we will save the seqeunce to save the steps later
        seq = []
        curr = m
        while True :
            # we see if we have already recorded steps for curr
            # note that dict.get(key) will return None if the key is
            # not in the dictionary
            curr_steps = known.get(curr)
            if curr_steps is not None :
                # we now know there are curr_steps more to go
                # so we don't need to waste our time computing
                m_steps += curr_steps
                # record all new steps counts for the sequence
                for idx,val in enumerate(seq) :
                    known[val] = m_steps - idx
                break
            seq.append(curr)
            if curr % 2 == 0 :
                curr = curr // 2
            else :
                curr = 3 * curr + 1
            m_steps += 1

        if m_steps > max_steps :
            max_steps = m_steps
            max_start = m

    return (max_start, max_steps)
```

- **b.** What is `collatz_max(1000000)`?

In [14]:

```
import collatz

print(collatz.collatz_max(10**6)) # takes a few seconds
print(collatz.collatz_max(10**7)) # takes about 30 seconds
```

```
(837799, 524)
(8400511, 685)
```

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.

In [15]:

```
import random

def satisfies_fermat(n) :
    """ Returns True if the positive integer n
        satisfies the conclusion of Fermat's little
        theorem. False otherwise. """
    if type(n) is not int or n < 0 :
        raise ValueError("Input should be a positive integer.")
    # return for all even numbers
    if n == 2 : return True
    if n % 2 == 0 : return False
    # make a randomized list of tests
    tests = list(range(2,n))
    random.shuffle(tests)
    for x in tests :
        if pow(x, n-1, n) != 1 :
            return False
    return True
```

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_pseuoprime(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.

In [21]:

```
import math

def num_tests(n,prob) :
    """ Returns a number of tests needed to
        achieve probablity prob or more for
        n to be a Fermat pseudoprime. """
    if prob >= 1. :
        # since half of the numbers should be Fermat
        # witnesses, we return 1+(n-1)//2
        return 1+(n-1)//2
    else :
        # return k such that 1/2^k <= 1 - prob
        return -math.floor(math.log2(1-prob))

def probably_fermat_pseduoprime(n, prob) :
    """ Returns True if the positive integer n
        has probability `prob` (or more) of being
        a Fermat pseudoprime . """
    if type(n) is not int or n < 0 :
        raise ValueError("Input should be a positive integer.")
    # return for all even numbers
    if n == 2 : return True
    if n % 2 == 0 : return False
    k = num_tests(n, prob)
    tests = random.sample(range(1,n),k)
    for a in tests :
        if pow(a, n-1, n) != 1 :
            return False
    return True
```