CoCoNuT Assignment One

January 20, 2015

1 Introduction To Sage

We start with a basic introduction to Sage. We introduce basic commands, and after which we will give some problems. All of the answers to the problems should make use *only* of the commands given in this section. The reason for this is that Sage is VERY powerful, and so one can actually solve most problems with a single command. We however want you to learn both *how* to use Sage, and *how* sage actually *works*.

Sage is basically Python, with a lot of mathematical software compiled in and available from a Python command prompt. Launch Sage with the command `sage` on a lab machine. It will display the prompt `sage:`, enter a command to get output on the line below[1].

Note: Any code you write directly into Sage is not saved, i.e. it will be deleted once you exit the session. A good idea is to save you code into a .sage file and then you can upload it using the command `load("filename.sage")`.

Variables and Arithmetic

```
sage: 1+1
2
```

Division with remainder uses the `//` and `%` operations.

```
sage: 5 // 3
1
sage: 5 % 3
2
```

You assign variables with the `=` sign. By default, `x` is a symbolic[2] variable, all other variables are unassigned. To make more symbolic variables, declare them with `var`.

```
sage: u=1
  (no output)
sage: u
1
sage: v
  (error - since v is not assigned)
sage: x
x
sage: x+x+1
```

[1] If this does not work on any CS dept machine, let Nigel know the machine name.

[2] That is, the value of `x` is an object that handles its own arithmetic.

```
2*x + 1
sage: var('y')
y
sage: x+y
x + y
```

Conditional Statements and Loops

If, for and while statements work as in Python (Sage *is* Python, after all). Sage will auto-indent lines (4 spaces) and display a `....:` prompt after an if/for/while statement, use backspace to enter elif/else clauses. Entering an empty line ends and evaluates the statement. The equality operator for numbers is `==` and the `range(n)` function returns an array of integers from 0 to $n-1$. You can also use `range(n, m)` to get integers from n to $m-1$ and `range(n, m, s)` to set a custom step.

```
sage: u=1
sage: v=2
sage: if u<v:
....:     print 'less than'
....: elif u == v:
....:     print 'equal'
....: else:
....:     print 'greater than'
....:
less than

sage: u=10
sage: v=1
sage: while u > v:
....:     print u
....:     u = u - 1
....:
10
9
8
7
6
5
4
3
2

sage: for u in range(5):
....:     print u
....:
0
1
2
3
4

sage: for u in range(1,10,3):
....:     print u
```

```
....:
1
4
7
```

Functions

Sage offers two types of functions, mathematical functions using symbolic variables (on which one can do calculus etc.) and regular Python procedures, introduced with the `def` keyword.

```
sage: f(x) = x + 1
sage: f(2)
3

sage: def g(x):
....:     if x == 0:
....:         return 1
....:     else:
....:         return x * g(x-1)
....:
sage: g(5)
120
```

Lists

The list data type in Sage stores elements of arbitrary type.

```
sage: L=[1,2,'A','B']
sage: L[0]
1

sage: L=[i for i in range (5)]
sage: L
[0, 1, 2, 3, 4]
```

Polynomials

Examples of defining polynomials in Sage:

- Integer coefficients: Univariate polynomials in x with integer coefficients can be define as follows: `sage: ZP.<x> = ZZ[]` Or `sage: ZP.<x> = Integers()[]` One can perform the usual arithmetic operations on polynomials in the same manner as one does with numbers.

  ```
  sage: p1= x^5 + 3*x^2 - 2*x + 7
  sage: p2= x^2 + x
  sage: p1*p2
  x^7 + x^6 + 3*x^4 + x^3 + 5*x^2 + 7*x
  sage: p1//p2
  x^3 - x^2 + x + 2
  ```

- Polynomials with coefficients in $\{0, \ldots, n-1\}$, i.e. the integers modulo n. Example for $n = 7$,

  ```
  sage: ZP.<x> = (Integers(7))[]
  ```

One can similarly define multivariate polynomials `sage: ZP.<x,y> = ZZ[]`

Primes

To check whether or not a variable/number is prime, use the function `is_prime()`. e.g. `is_prime(11)` will return True. To get the smallest prime $> n$, use the `next_prime(n)`. e.g. `next_prime(11)` will return 13. Similarly, `previous_prime(n)` returns the largest prime that is $< n$. The function `prime_range(a,b)` returns a list of the primes which are $\geq a$ and $< b$.

2 Assignment One Questions

1. (a) Using *only* the techniques discussed above, implement a function in sage called `MyPowMod(a, b, c)` that takes three integers a, b, c as input and returns $a^b \bmod c$.

   Answer:

```
def MyPowMod(a, b, c):
    x = 1
    while b > 0:
        if b % 2:
            x = (x * a) % c
        a = (a * a) % c
        b = b // 2
    return x
```

   (b) Use your function to compute 5385892759875 ^ 409784891274 (where ^ denotes exponentiation) mod 5427528967528756.

   Answer:

   `304633414115229`

2. (a) Again using only the techniques above, write a function `MyGCD(a,b)` that computes the greatest common divisor of two integers.

   Answer:

```
def MyGCD(a,b):
    b=abs(b)
    while b<>0:
        r= a % b
        a= b
        b= r
    return a
```

(b) Compute the GCD of 593085902352 and 8752389742891 using your function.

Answer:

```
1
```

3. (a) Again, using the above techniques only, write a function `MyLCM(a, b)` that computes the least common multiple of a and b.

Answer:

```
def MyLCM(a,b):
    a=abs(a)
    b=abs(b)
    return (a*b)//MyGCD(a,b)
```

(b) Compute the LCM of 55902352 and 8381902352 using your function.

Answer:

```
29285503481945744
```

4. In this question your answers should be the sage code needed to produce the answer, and not the specific answer (which is trivial).

(a) Create a list L containing the odd integers between 1 and 1911.

Answer:

```
L=range(1,1912,2)
```

(b) Reverse the order of the items in L.

Answer:

```
L.reverse()
```

(c) Compute the number of elements in L?

Answer:

```
len(L)
```

(d) Append the values 7,19 to L.

Answer:

```
L=L+[7,19]
```

(e) Convert the List L into a set S.

Answer:

```
S=Set(L)
```

(f) What is the cardinality of S?

Answer:

```
S.cardinality()
```

5. The extended GCD algorithm is an extension of the GCD algorithm which besides computing the GCD of a and b, it also finds the integers x and y satisfying $x \cdot a + y \cdot b = \text{GCD(a,b)}$. The Sage command `xgcd(a,b)` will return a list of 3 elements (GCD(a,b),x,y) satisfying the above equation.

```
sage: xgcd(12,15)
(3, -1, 1)
```

(a) Using your prior knowledge (or Wikipedia) write a Sage function `MyXGCD(a,b)` that mimics the inbuilt `xgcd(a,b)` command.

Answer:

```
def MyXGCD(a,b):
  s = 0
  old_s = 1
  t = 1
  old_t = 0
  r = b
  old_r = a
  while r <> 0:
    q = old_r // r
    (old_r,r) = (r,old_r-q*r)
    (old_s,s) = (s,old_s-q*s)
    (old_t,t) = (t,old_t-q*t)
  return [old_r,old_s,old_t]
```

(b) Using only the commands above, write a function `Findx` which on input a and b outputs x satisfying $a \cdot x = 1 \pmod{b}$ if such x exists. If such a value does not exist, your function must display an appropriate message.

Answer:

```
def Findx(a,b):
       (g,x,y) = xgcd(a,b)
       if(g == 1):
         return x % b
       else:
         print "x does not exist"
```

6. Using only the commands above, write a function `MyFactor` which on input a large integer n returns its prime divisors and their exponents. e.g. `MyFactor(18)` will return [[2,1], [3,2]]

Answer:

```
def MyFactor(n):
   if is_prime(n) or n==1: return [n]
   currentprime = 2
   lastprime = previous_prime(int(n^0.5) + 1)
   factors = []
   while currentprime  <= lastprime:
       if currentprime*currentprime > n: break
       pcount = 0
       while n % currentprime == 0:
           pcount = pcount + 1
           n =  n // currentprime
       if pcount > 0: factors.append([currentprime,pcount])
```

```
    currentprime = next_prime(currentprime)
if n > 1: factors.append([n,1])
return factors
```