Math 56 Compu & Expt Math, Spring 2013: Homework 7

due 10am Tuesday May 21st

Last one! Shorter one to let you focus on projects. The last question has a little more coding, but I give you lots of clues, and encourage collaboration.

1. BBP algorithm for arbitrary binary digits of $\ln 2$.
   (a) Prove that $\ln 2 = \sum_{k=1}^{\infty}(1/k2^k)$. [Hint: take $\frac{1}{1-x^{-1}} - 1 - \frac{1}{x}$, expand, and integrate over $(a, +\infty)$, for some a, in two different ways.]
   (b) Write a Matlab (using standard doubles to represent integers), or python (not arbitrary precision) function `r = expmodk(b,n,k)` that implements fast binary exponentiation (mod k) to compute $r = b^n(\text{mod}k)$. Include a driver that tests it on small numbers for which the answer can be computed directly, including annoying cases like $n = 0$, $k = 1$, etc.
   (c) Use this to code up the formula for $\{2^d \ln 2\}$ using the two sums. You'll want Matlab's `mod(x,1)` or python's `x % 1` applied to *each* term in the 1st sum. Include 50 or so terms in the exponentially convergent 2nd sum. Output the fractional part in binary, e.g. using `dec2bin` in Matlab. [Hint: test your code by changing d by e.g. 20 and checking that the overlapping digits are the same.]
   (d) Compute the 50 binary digits starting at the 10^7th, for $\ln 2$, and state how long your code took to run. [Hint: warm up on smaller d values, and make sure to check the last digits using the above overlapping method.]

2. In the file `threekeys.txt` are three 1024-bit RSA public keys (numbers of size around 2^{1024} of the form $N = qp$ with p, q prime), in decimal. Your goal is to crack (factor) them. You'll want to work in sage/python.
   (a) It turns out that two of them have a factor in common. Find it, and hence crack two of the three, i.e. give the factors. (This is a toy version of the following: by mining thousands of RSA keys for common factors it was discovered in February 2012 that, due to faulty random number generator, are way more common than expected!)
   (b) Factor the remaining key by writing a short code for Fermat's method. How many steps did you need? (This shows the danger of having $q - p$ not much bigger than $\sqrt{p}$.)

3. Build a python function for Kraitchik's algorithm from class; this is a baby "Quadratic Sieve" (without an actual sieve). This will be good practise handling python *lists*, and satisfying when done. Your interface should be `kraitchik(N,y,r)` where N is the integer to factor, y sets the maximum size of prime to include in your factor base, and r is the number of successive x values to try. Use your code to factor the numbers:
   (a) 1180591624032052314157 (factors are too far apart for Fermat, way too large for trial division)
   (b) $2^{2^6} + 1 = 18446744073709551617$, the 6th Fermat number (although more specialized methods exist for Fermat numbers. . .)

   Tuning the parameters: increase r until you get around as many y-smooth candidates $x^2 - N$ as the size of your factor base. Increase y a bunch then repeat. Eventually you'll start finding useful vectors in the kernel. (a) and (b) shouldn't exceed 1 minute runtime (seek help if they do).

   Hints:

- I will let you use `fb = prime_range(y)` to set the factor base
- I will let you use the built-in `f = factor(...)` to extract the small prime factors of the "small" (compared to N) numbers $x^2 - N$. This is wasteful but saves you coding a sieve (or batch trial division).
- To build a matrix over $\{0, 1\}$ from a list of lists, eg, `a = [[1,2],[3,4],[5,6]]`, use `A = matrix(GF(2),3,2,a)`
- To find the set of null-space row vectors use `A.left_kernel().basis()`. Each vector is a *tuple*.
- Python tricks: elementwise eg multiplication of two lists `x` and `y` done by: `[a*b for a,b in zip(x,y)]`. You will only need this if you decide to construct v from its prime factors[1]. Product of a list is done by `prod`. Add `y` to a list `x` via `x.append(y)`. Ask if stuck.
- Here's a routine that does the messy job of converting the output of `f=factor(...)` into a list of exponents (0,1,2, etc) given `fb` the factor base list:

```
def factorbaseexponents(fb,f):
    """convert factor tuple list f into a list of factor base exponents
    for the factor base list fb
    """
    fbex = [0]*len(fb)        # list of zeros
    for t in list(f):         # loop over tuples in factors
        i = fb.index(t[0])    # 1st el of tuple has to be in fb list since smooth
        fbex[i] = t[1]
    return fbex
```

- Debug your code on the worksheet example, printing out everything. Then debug on `kraitchik(1098413,25,200)`, which should find the six x values 1051, 1063, 1077, 1119, 1142, 1237 for which $x^2 - N$ is 25-smooth, and a kernel vector (0, 1, 1, 0, 0, 1) which leads to the factorization 563×1951. The latter example comes from Brent's 2010 slides on the course website.

[1] Actually this is more efficient for huge cases—see Crandall–Pomerance book p.268 (6)—but you'll find it easier to get v from $\sqrt{(x_1^2 - N)(x_2^2 - N)\cdots}$