# Math 56 Compu & Expt Math, Spring 2013: HW7 Debriefing

1. 4+3+3+3 = 13 pts

   (a) Given the hint, this was routine. Eg see Tom. Term-wise integration of the sum was justified since for all needed $x \geq 2$ the sum is absolutely convergent. Notice how I cancelled off the 0th and 1st term so that all remaining integrals are finite.

   (b) Keep in mind that the line `t = 2^floor(log(n)/log(2));` from my conversion-to-binary code is not robust, since if the floating-point calculation of the logs happens to be slightly under the Better is to count $t$ up from 1 via a little loop of repeated doubling.

   Ben had neat idea of using `bin2dec(n)` to generate a string which was read off to decide whether factors of $b$ appear at each squaring. Kunyi did same neat idea using python's `bin`.

   Taking into account the "annoying" cases is best done by hand at the top of the code, eg see John. Hanh included a negative $n$ sanity check. You lost a point if the annoying case $n = 0$,$k = 1$ incorrectly gave $r = 1$. Fortunately this case doesn't arise in the BBP sum.

   (c) Output 50 digits of the fractional part in binary via `dec2bin(x * 2^50)` in Matlab.

   Kunyi did it all in python, which is useful.

   (d) At $d = 10^7$ this takes about a minute per run on a single core of a modern CPU such as i7, but only gets you around 41 digits correct (due to cumulative rounding errors in the large number $d$ of floating-point additions). You should have noticed due to the hint from part (c). To get 50 correct digits you need to also use digits from a 2nd run at eg $d = 10^7 + 20$. The correct digits are then

   10111001011001100101011000110101100011111101111110

   Eg see Kyutae or Kunyi.

2. 5+4 = 9 pts

   Read in the numbers via eg `f=open("threekeys.txt")` then doing `f.readline()` a few times.

   Everything is integer operations, so you don't need `mpf` or `mpmath` library any more—that was for floating point.

   (a) Notice you have to do GCD on all pairs. This would be $O(n^2)$ for $n$ public keys (here $n = 3$). However, amazingly, Dan Bernstein invented a *batch GCD* that lets you do this in $O(n \ln n)$, enabling common factors in large numbers of keys to be found.

   I gave +1 bonus for writing own GCD (I expected you to use sage's).

   Kyutae discovered that Fermat can factor all three keys I gave you, the first two in under 2 million steps. (Oops! But gcd is still much quicker)

   (b) Fermat takes only around 2 million steps, a few seconds if you use integer arithmetic and `is_square()` to test for a perfect square. It's best *not* to use `mp.dps` (arb prec floating-point), since sage already has arbitrary-precision *integers* built in.

   The danger of such an easy Fermat-crackable key can be averted by having $q - p$ much bigger than $\sqrt{p}$.

3. 10 pts

   This was the most elaborate bit of coding yet; a nice finale! One reason was that some of you got an overflow (infinity) if you computed $v$ via the sqrt of $(x_1^2 - N)(x_2^2 - N)\cdots$, so had to product up the individual prime powers to get $v$.

Both take around 10 seconds to factor, and need a factor base of at least primes up to 3000, and around 50000–100000 $x$ values.

The Fermat number $F_6$ in (b) was factored by Landry in 1880.

Ben had great use of code comments in Kraitchik.sage. Learn from this!