Assignment description

For this assignment, use the FFT code to do large integer multiplication.

Given two positive integers, represented in base 2^32 (a list of python short ints (32 bit), each representing a unique digit in our base 2^32 number representation), find the product and print out the result in the same representation.

As we discussed in class, we will use the evaluation,  multiply and interpolate method.

Given N0 and N1 as the input numbers, each of length n (a power of two):

1) pad N0 and N1 with n zeros in the high-order positions (why do we do this?)

2) evaluate N0 and N1 at 2n primitive roots of unity (use the forward FFT),

                      S0 <-- FFT(N0+pad),  S1 <-- FFT(N0+pad)

3) element-wise multiply the output of the evaluations of N0 and N1,  S <-- S0[i]\*S1[i], 0<=i < 2\*n

4) interpolate using the inverse FFT

5) extract the real coefficients and round each to integers

To Do, document, and hand in:

1. Run the system on some smaller problems and compare the solutions to the school algorithm to verify correctness. (5 points)
2. Notice how the solutions are always a bit "off" in the smallest significant digits? This is due to us having to "slice the pie" so thin (cram in many roots of unity in the unit circle) as n gets large. To measure this inaccuracy, construct a plot of average absolute error (the difference between the correct answer and the real part of the solution). Just sum the absolute differences and then divide by 2n for each problem size. Plot this as a log (x) linear (y) graph. Does this go up with n? Will this inaccuracy ever result in an error in the **integer solution?** Think about it. (10 points)
3. Repeat the timing experiment that we did for the two previous methods (school book and 3subproblem) for the FFT method. Plot a graph showing all three methods, where each method's results line goes up to a maximum run time per problem (say around 20 minutes of CPU time). At this run time limit, what are the problem sizes that can be solved by the three methods? (15 points)
4. For some selected problems printout the inputs and outputs in the correct base 4294967296 format, defined as a string of "digits" that range from "0000000000" to "4294967295"  highest order first, separated by "," (10 points)
5. Documented code (20 points)

For example,

0000889488,0000058477,0000000032,0000896802,0003373494,0000000025,0018551509,0000254364,0000000401,0000000008

is a legal printout of our big int representation. However

0000000131,0000156503,0000000007,0000390638,6000000231,0000032509,0000169717,8742924828,0000229691,0060905748

is not. Why?

For this last part, you will have to fix the numbers that come back from the polynomial multiply. Some of the numbers returned may overflow our base representation since they could be greater or equal to 4294967296 . Just as if the result of an addition or multiplication in base 10 produces an answer greater than 9, the additional amount will need to be "carried over" to the next highest digit. This process may need to be repeated for each digit, starting at the least significance. Make sure you document your solution code for this step.