ASSIGNMENT 3: SOLVING CONGRUENCES
DUE WEDNESDAY, NOVEMBER 30, 2011 AT 11:59PM

Contents

For this assignment, you will need a functioning gcd algorithm, Bezout algorithm, and possibly a factorization algorithm.

1. Linear congruences

We start by writing a handy function for much of what follows.

Problem 1 (5 points)**.** Write a function, modinverse(a, n), which given a positive integer n and an integer a, finds the multiplicative inverse of $a \mod n$, if it exists. More precisely, if this inverse exists, the function should return the integer b with $0 \leq b < n$, such that $ab \equiv 1 \bmod n$, and if the inverse does not exist, the function should return the 'None' type. The function should run nearly instantaneously even if a, n have hundreds of digits in them. (You type 'return None' to do this.)

Make sure this function works, because you will probably want to use it in several of the following problems!

The next three problems provide a complete implementation of a Chinese Remainder Theorem algorithm.

Problem 2 (10 points)**.** Write a function, simplecrt(a, n), which given two lists $a = [a_1, \ldots, a_k], n = [n_1, \ldots, n_k]$ of identical length, where the n_i are mutually coprime positive integers and the a_i arbitrary integers, finds the unique solution to the system of linear congruences $x \equiv a_i \bmod n_i$, for $1 \leq i \leq k$. More precisely, the function returns a tuple x, n, where $x \bmod n$ is the set of solutions to this system, with x the smallest non-negative solution.

The function should run nearly instantaneously even for data consisting of hundreds of digits of information. You may assume that the n_i you are given in this function are mutually coprime.

Because of the speed requirement of the previous problem, the trial-and-error method you use to solve simultaneous systems by hand is not going to work for all test cases. Instead, you should look at the proof of the CRT, which provides an explicit construction of the unique solution.

Problem 3 (10 points)**.** Write a function, compatibility(p, a, e), which given a prime number p, and two lists $a = [a_1, \ldots a_r], e = [e_1, \ldots, e_r]$, where the a_i are arbitrary integers and the e_i positive integers, determines whether there is a solution to the system of linear congruences $x \equiv a_i \bmod p^{e_i}$. More precisely, if there is a solution, the function should return the smallest non-negative solution, and if not, the function should return the None type. The function should run more or less instantaneously even for data consisting of hundreds of digits of information.

Be aware that the e_i are not necessarily sorted in increasing order!

Problem 4 (5 points)**.** Write a function, lcm(n), which given a list $n = [n_1, \ldots, n_r]$ of positive integers, returns $\mathrm{lcm}(n_1, \ldots, n_r)$, the least common multiple of the n_i. The function should run more or less instantaneously for input data consisting of several hundred digits.

Problem 5 (10 points)**.** Write a function, crt(a, n), which given two lists $a = [a_1, \ldots, a_r], n = [n_1, \ldots, n_r]$, where the a_i are arbitrary integers and the n_i are arbitrary positive integers, which tries to find solutions to the system $x \equiv a_i \bmod n_i$. More precisely, if there is a solution, the function should return a tuple x, n, where $x \bmod n$ is the unique solution to the system $x \equiv a_i \bmod n_i$, with x the smallest non-negative solution. If there is no simultaneous solution, the function should return 'None'. The function should run more or less instantaneously for n_i of size about 10 digits.

You can get up to 3 points of extra credit if you can write a version of this function which can handle input data of hundreds of digits in length almost instantaneously.

2. Polynomial congruences

There are a variety of ways to represent a polynomial in a computer, but for our purposes we will use one of the simplest methods: we will represent a polynomial $a_n x^n + \ldots + a_0$ by a list $[a_0, a_1, \ldots, a_n]$. For example, we will represent $f(x) = x^2 + 3x - 7$ by the list $[-7, 3, 1]$, and the polynomial $f(x) = 3$ by the list $[3]$.

Problem 6 (10 points)**.** Write a function, hensel(p, e, x, f), which given a polynomial f represented as a list $[a_0, \ldots, a_n]$, a solution x to $f(x) \equiv 0 \bmod p^e$, a prime number p, and a positive exponent e, finds all lifts of $x \bmod p^e$ which solve $f(x) \equiv 0 \bmod p^{e+1}$ using Hensel's Lemma. More specifically, the function should return None if there are no lifts of $x \bmod p^e$ which solve $f(x) \equiv 0 \bmod p^{e+1}$, and otherwise should return a list of all lifts which are solutions. (By Hensel's Lemma, there should be either 1 or p elements in this list.) The function should run more or less instantaneously even if there are hundreds of digits in the input.

In the process of solving the above problem you will probably want to look at the proof of Hensel's Lemma in the case where $p \nmid f'(x)$ to find a constructive method for finding the unique lift which solves $f(x) \equiv 0 \bmod p^{e+1}$ without using trial-and-error.

As the previous problem makes clear, if you are working with polynomials on a computer, you will almost certainly have to evaluate that polynomial at various points $x = a$. For example, suppose $f(x) = a_n x^n + \ldots + a_1 x + a_0$ is a polynomial with integer coefficients, and you want to evaluate $f(x)$ at the integer a. How many multiplications does a naive answer to this question take?

Well, you will calculate a^k for each k with $0 \leq k \leq n$; if you compute a^k using $k-1$ multiplications, then altogether it will take $1+2+\ldots+(n-1) = n(n-1)/2$ multiplications to find the powers of a, followed by n multiplications to compute the products $a_k a^k$. However, you might notice that you are doing lots of repeated calculations with this method; maybe you just compute each of $a, a^2, \ldots, a^k$ once: you start by calculating a_0, then by adding $a_1 \cdot a$ to a_0 (which takes 1 multiplication), and then adding $a_2 \cdot a^2$ to this, which takes 2 multiplications (one to compute a^2 from a, and another to compute $a_2 \cdot a^2$). You can repeat this procedure, and altogether this takes $2n-1$ multiplications to evaluate $f(a)$.

There is actually an even better method, known as Horner's algorithm: notice that we can write

$$f(a) = a_0 + a_1 a + \ldots + a_n a^n = a_0 + a(a_1 + a_2 a + a_3 a^2 + \ldots + a_n a^{n-1}).$$

If you have already computed the polynomial in the parentheses $(a_1 + a_2 a + \ldots + a_n a^{n-1})$, then it only takes one multiplication and one addition to compute $f(a)$. But notice that the polynomial in the parentheses is a degree $n-1$ polynomial. So we can apply this method to the $n-1$ degree polynomial; in general we can reduce the degree of the polynomial we have to evaluate in exchange for one addition and one multiplication. Altogether, we will need a total of n multiplications to evaluate $f(a)$.

A good exercise is to implement this algorithm on your computer. You don't need to turn this in, but it is a nice exercise. Try implementing a version which uses recursion and another version which is iterative.

3. Useful information

3.1. **Problems to hand in.** In your file which contains the code you submit, make sure that you can run the file in IDLE without error and that all functions are loaded appropriately. In other words, if you open the .py file you submit and run it in IDLE (using the Run Module command under the Run submenu), there should be no error messages in the interactive interpreter, and you should be able to use the functions you wrote. Also make sure that you spell the names of the required functions correctly! (On the other hand, the names of the arguments passed into the function can be arbitrary, as long as the function accepts the correct type and number of arguments.) Finally, please put your name near the top of the file in a comment (use the # symbol to write comments; Python will ignore everything on that line which comes after the #).

- Turn in problems 1, 2, 3, 4, 5, and 6 in a single file named [lastname]4.py (without brackets around your last name.)

3.2. **Test cases.** (This is simply a repeat of the information provided in the previous assignment.) The Python standard library contains a handy module called doctest, which provides a lightweight method for testing test cases. In the assignment page, the template .py file will contain a lengthy *docstring* at the beginning of each function, which contains the test cases we provide for each function (if there are any).

Also, at the end of the file, there will be a short snippet of code, which will only run if you ask IDLE to execute the entire file with the 'Run Module' command. The template we provide will include a 'verbose=True' switch, which will cause your

program to output extensive information on the results of each test case. If you would like to suppress the output, and only be informed if cases fail, remove 'verbose=True' from within the parentheses of the doctest.testmod function.

For questions that use floating point numbers, we only test whether your answer is within some degree of accuracy. It is conceivable for your function to fail those testcases even if it is correct because of the way floating point arithmetic is handled on the machine being used.

Usage of the test cases is optional. If you want, you can delete the docstrings, delete the code at the end which calls doctest.testmod, or just not use the template file provided. However, we strongly recommend you test your functions against some test cases to ensure that your code is at least somewhat functional.