Homework 5

Due Thursday, October 8

Implementation Part

1. Implement Sun Tzu's algorithm for solving concurrent congruences (i.e., the Chinese Remainder Theorem). Specifically, define a function `CRT(moduli,residues)` which satisfies the following:

Input	Output
A list of moduli $m_1, \cdots, m_t$ (positive integers) A list of integers $a_1, \cdots, a_t$	If moduli are pairwise coprime x satisfying $x \equiv a_i \mod m_i$ for all i. Otherwise an error message.

Hints:

- One way you could do this is to make an auxiliary function `CRTPairs(m1,m2,a1,a2)` which solves the problem for 2 congruences, and have `CRT` feed recursively into `CRTPairs`
- Naively checking the moduli are coprime takes running the Euclidean algorithm $\mathcal{O}(t^2)$ times, but you should be able to do so only running it $\mathcal{O}(t)$ times.

2. Implement the baby steps-giant steps algorithm to solve the DLP for $\mathbb{F}_p^*$. In particular, define a function `babyGiant(g,h,p,N=p-1)` satisfying the following:

Input	Output
A prime p	$\log_g(h)$ if it exists
An element $g \in \mathbb{F}_p^*$	
An element $h \in \mathbb{F}_p^*$	
The order N of g.	

If no order N is given your algorithm should set $N = p - 1$.
Hint

- Recall that the hardest part of this algorithm wasn't making the lists of baby steps and giant steps, but of searching for and finding an element that is in both lists. If you do this by just comparing each element of the list one by one, then this will take $\mathcal{O}(\sqrt{N}^2) = \mathcal{O}(N)$ steps, and you won't have saved any time at all over the brute force attack of the discrete log. Instead, we will implement a *hash table*. In python this type of data structure is a *set*. You could initiate this like `babysteps = set()`, and then add an element x using `babysteps.add(x)`. The great thing about sets is you can do something like `x in babysteps` which will return true if x is in the set and false otherwise, and since x is paired with an index it can do this in $\mathcal{O}(1)$ time! The downside is hash tables aren't ordered, so you need some other way remember the discrete logs of your elements. One way is to also have a babysteps list alongside the set, and each time you generate a giant step, see if it is in the set (using `x in babystep` which is $\mathcal{O}(1)$), if it is then run through the babystep list to see which position the match is. This is better because you are only running through the babysteps list once (rather than $\sqrt{N}$ times with the naive list comparison).

3. Implement to Pohlig-Hellman algorithm to solve the DLP for an element $g \in \mathbb{F}_p^*$ of order $N = m_1 m_2 ... m_t$ (for coprime m_i). Specifically, define a function `pohligHellman(g,h,p,factors)`

Input	Output
A prime p	$\log_g(h)$ if it exists
An element $g \in \mathbb{F}_p^*$	
An element $h \in \mathbb{F}_p^*$	
The prime power factors $m_1, \cdots m_t$ of $\lvert g \rvert$	

Hints

- The structure should loosely be as follows. Reduce the problem to solving the DLP for elements of smaller order, let `babyGiant` solve those problems (make sure to tell it the order is smaller, otherwise you aren't saving any time), and then use `CRT` to stitch them together.
- It is difficult in general to compute $|g|$ (about as difficult as factoring $p-1$), and so checking if the m_i are indeed the prime factors of $|g|$ may be difficult. Instead, check that $g^{m_1 m_2 ... m_t} = 1$. In this case your algorithm should still work (see Problem 6).

4. Use `CRT` to solve to following sets of congruences, and check that the solution given works.

   (a) $x \equiv 9 \mod 23$ and $x = 25 \mod 41$

   (b)

$$\begin{aligned} x &\equiv 1 \mod 2 \\ x &\equiv 2 \mod 3 \\ x &\equiv 4 \mod 5 \\ x &\equiv 6 \mod 7 \\ x &\equiv 10 \mod 11 \\ x &\equiv 1 \mod 13 \\ x &\equiv 16 \mod 17 \end{aligned}$$

5. Let's test out your log functions!

   (a) Let $p = 113$. Use baby steps-giant steps to compute $\log_3 19$ modulo p.

   (b) Notice that 112 factors as $2^4 * 7$. Use this information and Pohlig-Hellman to compute $\log_3 19$ and see if your answer matches.

   (c) Let $p = 30235367134636331149$. Try using baby steps-giant steps to compute the discrete log $\log_6 3295$ modulo p.

   (d) You might have had trouble getting the last one to run. I did. What if I told you that $p-1$ has the following prime factorization?

$$p - 1 = 2^2 * 3^2 * 13 * 41143 * 335341 * 4682597.$$

   Now use Pohlig-Hellman to speed up your computation. (It speeds it up considerably!). Use fast powering to make sure you got the right answer (it is very satisfying!).

Written Part

6. In defining `babyGiant` and `pohligHellman` we didn't always know the exact order of g, and couldn't necessarily check it directly. Therefore the exact proofs of correctness of these algorithms in class need to be modified to make sure our algorithms work. Let's do this here.

   (a) If `babyGiant` doesn't receive an order as an input it defaults to $|g| = p-1$. Explain why this is just assuming that g is a primitive root. Suppose g is not a primitive root but `babyGiant` still assumes $N = p-1$. Prove that `babyGiant` returns the correct logarithm.

   (b) More generally, suppose you specify a N as a positive multiple of $|g|$ in `babyGiant`. Prove that it returns the correct logarithm.

   (c) For `pohligHellman` instead of checking that the m_i were indeed the prime power factors of $|g|$, we just checked that $g^{m_1 m_2 \cdots m_t} = 1$. Prove that if this condition holds (and the m_i are still coprime) that `pohligHellman` returns the correct logarithm.

   (d) Even if the order given as input is not entirely correct (but say a multiple of the actual order), these algorithms work. Still, there are drawbacks to not having the correct order. What are they? (e.g., if using $p-1$ in `babyGiant` always works, why don't we just always set $N = p - 1$?).

7. (a) Show that `CRT` runs in $\mathcal{O}(\log N)$ steps where $N = m_1 m_2 \cdots m_t$ is the product of the moduli. (You may assume your basic operations $+, -, \times, \div$ are all $\mathcal{O}(1)$. It is also worth pointing out that the number of prime of factors of N grows on average with $\mathcal{O}(\log(\log N))$, which is very slow. Since this is a rough upper bound for t, you may treat that as negligable, or instead show that CRT runs in $\mathcal{O}(\log N \log(\log(N))$.

   (b) Baby steps-giant steps as we introduced in class took $\mathcal{O}(\sqrt{N} \log N)$ because of how long it took to find an element in both the baby steps list and the giant steps list. In question 2 we implemented a hash table instead of a list, and checking if an element is in a hash table takes $\mathcal{O}(1)$ steps (which doesn't depend on the size of the table!). Use this fact to prove that your implementation runs in $\mathcal{O}(\sqrt{N})$ steps instead.

8. Let's prove the uniqueness part of the Chinese Remainder theorem.

   (a) Let a, b, c be positive integers and suppose that:

   $$a|c, \quad b|c, \quad \gcd(a, b) = 1.$$

   Then $ab|c$.

   (b) Suppose $m_1, \cdots, m_t$ are pairwise coprime positive integers, and suppose $a_1, \cdots, a_t \in \mathbb{Z}$. Show that if y and z are both solutions to the system of congruences

   $$\begin{aligned} x &\equiv a_1 \mod m_1 \\ x &\equiv a_2 \mod m_2 \\ &\vdots \\ x &\equiv a_t \mod m_t, \end{aligned}$$

   then $y \equiv z \mod m_1 m_2 \ldots m_t$

Let's finish by proving the following theorem:

Theorem 1. *Let m be an odd number and a an integer not divisble by any of the prime factors of m. Then a has a square root mod m if and only if $a^{\frac{p-1}{2}} \equiv 1 \mod p$ for every prime factor p of m.*

9. (a) Let a be an integer not divisble by an odd prime p. Show that a has a square root mod p if and only if $a^{\frac{p-1}{2}} \equiv 1 \mod p$. (*Hint:* Use HW2 Problem 8.)

   (b) Let $m = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_t^{\alpha_t}$ and a an integer. Show that a has a square root mod m if and only if it has a square root mod $p_i^{\alpha_i}$ for each i. (*Hint:* Use the Chinese Remainder Theorem.)

   (c) Let m be an odd number and suppose a is an integer not divisible by any prime factor of m. Show a has a square root mod m if and only if it has a square root mod p for every prime p. (*Hint:* Use HW3 Problem 7).

   (d) Deduce Theorem 1 from parts (a),(b), and (c) above.

   (e) Can you relax any of the hypotheses of Theorem 1? For example, what if m is even? Or what if some prime factor of m divides a? Compute some examples and informally discuss your thoughts.