CS 70 Discrete Mathematics for CS — Spring 2008 — David Wagner — HW 5

Due Thursday, February 28th

1. (7 pts.) Practice with modular arithmetic

You don't need to justify your answers or show your work for any of the following parts of this question.

1. Suppose $3x+2y \equiv 3 \pmod{57}$ and $x+y \equiv 2 \pmod{57}$. Solve for x.
2. Suppose $x-y \equiv 20 \pmod{2008}$ and $x+y \equiv 8 \pmod{2008}$. List all the possible values for $x^2 - y^2 \bmod 2008$.
3. Suppose we know that $x \equiv 52 \pmod{63}$. List all the possible values of $x \bmod 7$.
4. Suppose we know that $x \equiv 52 \pmod{64}$. List all the possible values of $x \bmod 7$.
5. Suppose $x^{2008}+x+1 \equiv 0 \pmod{3}$ and $y \equiv x^{2008}+2008x+1 \pmod{3}$. List all the possible values of $y \bmod 3$.
6. Suppose $3x \equiv 1 \pmod{16}$. List all possible values of $x \bmod 16$.
7. Suppose that $x+5 \equiv 10 \pmod{64}$. Are we guaranteed that $x \equiv 5 \pmod{64}$?

2. (8 pts.) More practice with modular arithmetic

1. Suppose that $2x \equiv 10 \pmod{64}$. Are we guaranteed that $x \equiv 5 \pmod{64}$? Briefly, why or why not?
2. Suppose that $3x \equiv 15 \pmod{64}$. Are we guaranteed that $x \equiv 5 \pmod{64}$? Briefly, why or why not?
3. Compute $\gcd(633, 153)$ using the Euclidean algorithm. Show your work (e.g., show the recursive calls made by the Euclidean algorithm).
4. Use the extended Euclidean algorithm to find some pair of integers $j,k \in \mathbb{Z}$ such that $52j + 15k = 1$. Show your work.

3. (6 pts.) Spaced-out numbers

If n is a number written in decimal notation, let $f(n)$ denote the number obtained by inserting two zero digits between every digit of n. For instance, $f(7) = 7$, $f(12) = 1002$, $f(371) = 3007001$, $f(80) = 8000$, and so on. Prove that $f(n) \equiv n \pmod{11}$ for every $n \in \mathbb{N}$.

Hint: Express n in the form $\sum_{i=0}^{k} a_i 10^i$. What is $f(n)$, in terms of these a_i's?

4. (6 pts.) Name that algorithm

Consider the following mystery algorithm, which accepts two natural numbers as input:

```
Mystery($a, b$):
1. Set $x := a$, $y := b$, $z := 0$.
2. While $y \neq 0$, do:
3.     // Invariant: ______________
4.     If $y$ is odd, then set $z := z + x$.
5.     Set $y := \lfloor y/2 \rfloor$.
6.     Set $x := x \times 2$.
7. Return $z$.
```

Answer the following two questions. You don't need to justify or prove your answers.

1. Concisely, what does Mystery(a,b) compute? (Ignore line 3.)
2. What is the strongest invariant you can give that will always hold every time execution reaches line 3? (Your answer will be a relationship that relates some function of a,b to some function of x, y, and z.)

5. (5 pts.) Diophantine equations

Given positive integers a,b,c, the goal is to find an integer solution (for x,y,z) to the equation $ax+by+cz = 1$. In other words, a,b,c are given and x,y,z are the unknowns.

Design an efficient algorithm to find such a solution, assuming that $\gcd(a,b) = \gcd(a,c) = \gcd(b,c) = 1$.

6. (18 pts.) Program checking: "Casting out p's"

You've bought a fast integer calculation library from Goofle, Inc. ("featuring patented addition technology!"), and holy cow, their code is *fast*! However, you're a little suspicious about whether Goofle's code is always returning the correct answer. They don't supply the source code to their library, so you have no way to check their algorithms directly.

Instead, you decide to sanity-check every result you get back from their library at run-time to give yourself a good chance of detecting any erroneous results. To this end, you're going to write a wrapper (e.g., `CheckedAdd()`) around their API (e.g., `GoofleAdd()`) to check Goofle's result—hopefully without incurring too much performance penalty.

1. As a warm-up, prove that, for every positive $D \in \mathbb{N}$, the number of distinct prime factors of D is at most $\lg D$.
2. Now, back to writing `CheckedAdd()`. Let's suppose you use the following algorithm:

```
CheckedAdd($m,n$):
// Given $m,n \in \mathbb{N}$, computes $m+n$ using Goofle's library and checks for errors.
1. Set $k :=$ GoofleAdd$(m,n)$.
2. Pick a random 64-bit prime $p$, i.e., choose uniformly at random among all primes
   satisfying $2^{63} < p < 2^{64}$.
3. Compute $m' := m \bmod p$, $n' := n \bmod p$, $k' := k \bmod p$.
4. If $k' \neq m' + n' \bmod p$, then signal an error and abort.
5. Return $k$.
```

   Assume that Steps 2–4 can be performed using a trusted library that is known to work correctly.

   Prove that if `GoofleAdd`(m,n) correctly computes $m+n$, then no error will be signaled.

3. Next, let's check whether errors in Goofle's library will be detected. Let k denote the result returned by `GoofleAdd`(m,n) in Step 2. Prove the following: If `GoofleAdd()` returns the wrong result, then with probability at least $1 - \frac{\lg D}{2^{57}}$, an error will be signaled, where $D = \max(k, m+n)$.

   Hint: Even if Goofle is trying to fool us maliciously, they still have to pick k before they have any idea what prime we'll choose. What is their best strategy for picking k erroneously to maximize the probability of avoiding detection?

   Hint: You may use, without proof, the fact that there are at least 2^{57} different 64-bit primes.

   Hint: For some fixed values of $m,n,k \in \mathbb{N}$ such that $k \neq m+n$, call a prime p *unlucky* if choosing it in line 2 will cause `CheckedAdd()` to fail to signal an error (i.e., executing lines 3 and 4 with these values of m,n,k will not signal an error). If you can show that there can be at most $\lg D$ unlucky primes for any particular choice of m,n,k, you'll be almost done.

4. Goofle's library also includes a function `GoofleMultiply`(m,n) that claims to efficiently compute $m \times n$, when given inputs $m,n \in \mathbb{N}$.

   Design an algorithm `CheckedMultiply`(m,n) such that: (1) When m,n are sufficiently large numbers, `CheckedMultiply`(m,n) runs almost as quickly as `GoofleMultiply`(m,n); (2) Similarly to `CheckedAdd()`, you have a good chance of detecting erroneous results from any multiplication computation; and (3) Similarly to `CheckedAdd()`, you never signal an error without justification.

   You may assume that you have a trusted library that can be used to pick a random 64-bit prime p, add modulo p, and multiply modulo p, correctly and in $\mathrm{O}(1)$ time. Also, you may assume that your trusted library can compute $n \bmod p$ correctly, in $\mathrm{O}(\lg n)$ time.

   You should show your algorithm, but you do not need to prove it correct or justify your design.