Discrete Math in CS	**Problem Set 4**
CS 280 Fall 2000	**Due September 29, 2000**

Reading: Rosen, Sections 2.3 – 2.5.

As in previous problem sets, some of the questions present "claims," and ask you to decide whether they are true or false. For such a question, you should do one of the following two things: (i) give a proof of the claim; or (ii) give a proof that the claim is false, by showing that its negation is true. (Using a counter-example or other means.)

---

(1) In one of the *Die Hard* movies, Bruce Willis and Samuel Jackson have to disarm a bomb using the old scouting trick of measuring exactly four gallons of water using only a three-gallon container and a five-gallon container.

How general a trick is this, really? Here's a model for this type of problem. Suppose we have two containers, one that can hold exactly p gallons of water and another that can hold exactly q gallons of water. We also have a giant *vat* which we'll model (unrealistically) as being able to hold more than n gallons, for any natural number n; and we have a garden hose, which we'll model (also unrealistically) as being able to dispense an unlimited amount of water on demand.

Our goal is to use the two containers to put exactly c gallons of water in the vat, for some value of c. Here are the allowable operations.

- *Fill*: We can pick one of the containers and completely fill it with water from the garden hose.
- *Transfer*: We can pick one of the containers, and pour its current contents into the other container until one of the following two things first happens: (i) the container being poured from becomes empty; or (ii) the container being poured into becomes full.
- *Add to Vat*: We can pick one of the containers and pour its current contents into the vat.
- *Remove from Vat*: We can pick one of the containers and scoop water from the vat into it, stopping when one of the following two things first happens: (i) the container being filled becomes full; or (ii) the vat becomes empty.
- *Dump*: We can pick one of the containers and dump its entire contents on the ground (thereby emptying it, but not into the vat).

Consider the following claim.

Claim: Suppose we are given a p-gallon container and and a q-gallon container, where p and q are distinct prime numbers. Let c be any natural number. Then there exists a sequence of operations of the types described above that starts with both containers and the vat empty, and ends with the vat containing exactly c gallons of water.

Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

(2) Recall the definition of the Fibonacci numbers from class and the book: $f_1 = 1$, $f_2 = 1$, and for $n > 2$, we define $f_n = f_{n-1} + f_{n-2}$.

Consider the following claim.

Claim: For every $n \geq 2$, the greatest common divisor of f_n and f_{n-1} is equal to 1.

Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

---

(3) In class, we showed that if x and y are positive natural numbers such that $gcd(x, y) = 1$, then there exist integers a and b so that $ax+by = 1$. We observed that for such an equation to hold, it must be that one of the coefficients a or b is positive, and the other is negative.

Prove the stronger statement that if x and y are positive natural numbers such that $gcd(x, y) = 1$, then there exist integers a and b so that $a > 0$, $b < 0$, and $ax + by = 1$. (In other words, we are able to force the positive coefficient to be the one that is multiplied by x, rather than the one that is multiplied by y.)

(4) In class, we also gave a fast recursive algorithm to compute $a^x \bmod n$ for natural number a, x, and n. It worked as follows:

Function Exponentiate(a, x, n)
  If $x = 1$ then return x.
  Else if x is odd then return

$$a * \texttt{Exponentiate}(a, x-1, n) \bmod n.$$

  Else (x is even) return

$$(\texttt{Exponentiate}(a, x/2, n))^2 \bmod n.$$

Let $f(x)$ denote the number of multiplications performed by the function Exponentiate(a, x, n) — i.e. when it is called with an exponent of x. Notice that squaring a number requires a single multiplication, and that when $x = 1$, the function Exponentiate(a, x, n) does not perform any multiplications.

Prove by induction that for all positive natural numbers x,

$$f(x) \leq 2 + 2\log_2 x.$$

We've talked about the distinction between *good* and *bad* algorithms for problems involving natural numbers. A *good* algorithm is one for which the number of basic operations on an input of size n can be bounded by a function proportional to $(\log n)^c$, for some fixed exponent c. (I.e. like $c = 1$ or $c = 2$.) A *bad* algorithm is one for which such a bound does not hold; intuitively, it's an algorithm for which the running time on input n scales more like n than like the number of digits in n.

Thus, we saw that the elementary-school algorithms for addition and multiplication are good algorithms. Much less obviously, Euclid's algorithm for greatest common divisors is a good algorithm; and Question (4) above establishes that the function `Exponentiate` is a good algorithm. On the other hand, to find the prime factors of a number n, the approach of trying all possible divisors is a bad algorithm; indeed, no good algorithm is known for factoring, and it's suspected that no good algorithm exists.

RSA can clearly be broken by a bad algorithm — simply factor the public key by brute force. The real question is, can it be broken by a good algorithm? Let's explore this idea in more detail ...

---

(5) The phone rings one day and you pick it up. The voice at the other end says, "Hello, is this Professor Rivest?" You meant to say "no," but for some reason you said "yes" instead; the next thing you know, people from this mysterious company arrange to fly you out to the West Coast to advise them on their security system.

"We have a system where we put 100 agents out in the field," they explain to you, in a sub-basement surrounded by electromagnetic shielding. "Each agent announces an RSA public key, and then they use the RSA cryptosystem to send messages to each other using these public keys. We're not concerned about whether the agents know each other's private keys or not — we're all on the same team, after all — but we're very worried about outsiders learning the private keys.

"Here's the basic code we planned to use to assign a public key to each agent:

```
Method I:
  For i = 1, 2, 3, ..., 100
    Generate a prime number p_i.
    Generate a prime number p'_i.
    Compute the public and private keys for agent i from p_i and p'_i.
  End
```

"After looking at this for a while, we thought: 'Isn't Method I sort of wasteful of primes?' So we came up with Method II:

```
Method II:
  First generate prime numbers p_1, p_2, ..., p_101
  For i = 1, 2, 3, ..., 100
    Compute the public and private keys for agent i from p_i and p_{i+1}.
  End
```

"Notice how we only use about half as many primes in Method II. But we're worried that Method II is somehow insecure — *that if our enemies knew we were using Method II to compute keys, they could compute the private keys from the public keys by a good algorithm.*"

So here's the question: show that in fact the company's fears are well-founded. Given a set of 100 public keys known to be generated by Method II, show how to compute the 100 corresponding private keys by a good algorithm.

(**Note.** The point is that, in trying to break the keys, you know that Method II was used, but you (obviously) don't know at the outset what the primes $p_1, p_2, \ldots, p_{101}$ are. Also, here is a crucial thing: your good algorithm is allowed to have a factor proportional to the number of agents in its running time, since this is simply the (small) constant 100.)

(6) Your discovery that Method II is insecure causes much uproar; but when this dies down, they remember another method for generating public keys they want to tell you about.

"Basically, here's how it goes. For each agent i, we start with a fresh prime number p_i. Now we need a second prime p_i' to complete the construction of the key. So we start at $1+p_i$, and test each natural number in order for primality until we first find one that is prime. We use this as the second prime p_i'. If we try $2\log_2 p_i$ numbers and none of them is prime, then we decide this is taking too long and start over with a new choice of p_i — but in practice this rarely happens.

"This describes the full method; but to be completely precise, here's a more formal description of it:"

```
Method III:
  For i = 1, 2, 3, ..., 100
    Generate a prime number p_i.
    Test each of p_i + 1, p_i + 2, p_i + 3, ..., p_i + (2 log_2 p_i) for primality.
    If the test reports that any of these numbers is prime then
      Let p_i' be the smallest one that is reported to be prime.
      Compute the public and private keys for agent i from p_i and p_i'.
    Else
      Repeat this process for agent i, starting with a different p_i.
    Endif
  End
```

Unfortunately, Method III is also insecure. Given a set of 100 public keys known to be generated by Method III, show how to compute the 100 corresponding private keys by a good algorithm.