Discrete Math in CS	**Problem Set 3**
CS 280 Fall 2005	**Due September 23, 2005**

Reading: Rosen, Sections 2.4-2.6.

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) (a) Show the sequence of recursive calls executed by Euclid's Algorithm as it computes the greatest common divisor of 60 and 42. Include in your answer the actual value of $d = gcd(60, 42)$.

(b) In class, we also showed that if x and y are positive natural numbers such that $gcd(x, y) = d$, then there exist integers a and b so that $ax + by = d$.

With $d = gcd(60, 42)$, give numbers a and b so that $60a + 42b = d$. You do not need to show the steps you use to compute a and b.

(2) Suppose we have a function $h(n)$ defined on the positive integers that has the following properties.

(i) $h(1) = 1$ and $h(2) = 1$.

(ii) For all natural numbers $n \geq 3$, we have $h(n) = h(n-1)^2 + h(n-2)$.

The function $h(\cdot)$ grows very quickly after the first few values; for example $h(1), h(2), h(3), h(4), h(5), h(6)$ are $1, 1, 2, 5, 27, 734$ respectively.

Consider the following claim.

> *Claim: For every $n \geq 2$, the greatest common divisor of $h(n)$ and $h(n-1)$ is equal to 1.*

Prove this claim by induction on n.

---

(3) 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 positive integer. 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.

---

(4) The folks at MyCrawl are back, hoping you can help with a bizarre performance problem they just started having.

Here's the background on the problem. As they collect news headlines, they want to know if they've seen the headline before or not. They don't want to have to consult all the headlines in their index to do this, so they're using a very primitive form of *hashing* — a topic that we'll explore in more detail later in the course.

Their scheme works as follows. They pick a number n, and maintain n virtual "buckets" of headlines, labeled $0, 1, 2, \ldots, n-1$. Each headline in their index belongs to exactly one of these buckets. When a new headline a comes in, they map it to a number $f(a)$ between 0 and $n-1$, and they check whether it's already in bucket $f(a)$. If it's not, they add it to

this bucket. (So if the same headline a had come in previously, it would already be sitting in bucket $f(a)$ when they go to check.)

This is the essence of hashing in general: you divide up your data so that you only have to check a small fraction of it in situations like this. The hope is that if your function f — the "hash function" — spreads things out nicely, then you're only consulting a $1/n$ fraction of your data when you look in a single bucket.

To make all this concrete, one needs to specify the function $f(\cdot)$. Here's how they're doing it at MyCrawl. (Despite the long description that follows for the sake of completeness, it's actually describing a very standard thing one might do. (Though normally you'd use a prespecified bit encoding like ascii rather than inventing your own ...))

- Each headline is in the following structured style. It begins with the acronym of the wire service that carried the headline, then has a colon and a space, then the text of the headline itself, and then it ends with the date in the following format: two characters for the day of the month, then the first three letters of the name of the month, then four characters for the year,[1] and then the first three letters of the day of the week. For example:

  `AP: Turing's biographer wanted Russell Crowe in biopic 16Sep2005Fri`

- They then convert the text into a sequence of numbers, by mapping each character to a number as follows. The letters 'a' through 'z' are mapped to the numbers 1 through 26 (ignoring whether they're upper-case or lower-case), the "space" between words is mapped to 27, the numbers 0 through 9 are mapped to 28 through 37, and all other characters (e.g. punctuation) are mapped to a single "special symbol," which is encoded as the number 38.

- They then convert each number to binary, using exactly six bits per number — inserting leading 0's if necessary. (So 'a' becomes 000001, 'b' becomes 000010, ..., space becomes 011011, and so forth.)

- These sequences of bits are then all concatenated together, causing the headline to be represented as a single bit string. So for example, the bit string corresponding to the headline above would begin

  $$000001010000100110011011\cdots$$

- Finally, one views this long bit string as a (very large) number M written in binary notation.

- The value $f(a)$ is defined simply to be $M \bmod n$.

Now, there's the issue of what to use as the value for n. The MyCrawl people had read somewhere that it was a good idea in practice to have n be a prime number. They weren't

[1] yes, they'll have a Y10K problem, but by then they hope to all have finally made money from MyCrawl and retired ...

exactly sure why, but they did it, choosing the prime number $n = 4099$, and things worked very nicely — profiling indicated that when they went to consult a bucket, it tended to have roughly $(\frac{1}{4099})^{\text{th}}$ of their headlines.

Then at some point, it's not clear how, someone accidentally changed the code so that n became 4096. Immediately, performance became much worse — as far as they could tell, every time they mapped a headline a to $f(a)$, and then consulted bucket $f(a)$, this bucket seemed to contain at least 10% of all their headlines.

This is where they were hoping you could help them out. It's no trouble for them to switch back to $n = 4099$, but in addition to doing this they want to know what went so wrong with $n = 4096$ — is there a simple explanation for it, or does it hint at some bug deep in the internals of their system?

Explain why they observed a big performance problem with $n = 4096$. Your explanation should be based on the information you have above; if you want, you can make very mild further assumptions about their system, but the fewer the better. Also, it's not necessary to explain why 4099 worked so well, just why 4096 worked so badly.)

---

(5) A famous unsolved problem in number theory goes as follows. Consider the following function $C(\cdot)$ defined on the positive integers:

$$C(n) = \begin{cases} n/2 & \text{if } n \text{ even} \\ 3n+1 & \text{if } n \text{ odd and } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

That is, if n is even we divide it by 2, and if n is odd, we multiply it by 3 and add 1.

We can apply the function $C(\cdot)$ repeatedly, starting from any n. For example, starting with $n = 3$, we first get 10, then 5, then 16, then 8, 4, 2, and 1.

Here's the question:

> *Open Question: Is it true that for any positive integer n, if we repeatedly apply the function $C(\cdot)$ starting from n, we eventually reach the number 1?*

Notice how simple it was to specify this question — all it takes are the rules of addition, multiplication, and division. Despite this, it's fair to say that people don't really have the faintest idea how to resolve it. (The mathematician Paul Erdos said about this question, "Mathematics is not yet ready for such problems.")

We're not going to try solving the problem here, of course, but there are some interesting issues surrounding it that we can think about. First, a bit of notation. For numbers n and k, we say that $C_k(n)$ is the result of applying the function $C(\cdot)$ for k times in succession, starting from n. (So $C_1(n)$ is the same as $C(n)$.)

Using this notation, we can now reformulate the open question very compactly.

> *Open Question (reformulated): Is it true that for every positive integer n, there exists a k so that $C_k(n) = 1$?*

Now, a question that initially seems weaker is whether, for every $n > 1$, repeated application of the function $C(\cdot)$ eventually gets you to a number that is less than n. This too is unresolved.

> *Open Question 2: Is it true that for every integer $n > 1$, there exists a k so that $C_k(n) < n$?*

In fact, this question is equivalent to the original one: the answer to Open Question 2 is "yes" if and only if the answer to the original Open Question is "yes." (You can think about why this is true; it's not part of the problem here, but it's a nice bit of practice in doing induction.)

Suppose we take Open Question 2 and reverse the order of the quantifiers. That is, while Open Question 2 whether for every $n > 1$ there is a k so that $C_k(n) < n$, we could ask even more strongly whether there exists a (slngle) value of k so that for all $n > 1$, we have $C_k(n) < n$. That is,

> *Question 3: Is it true that there exists a k so that for every integer $n > 1$, we have $C_k(n) < n$?*

Notice the difference in the new question: it asks whether there's some fixed number of steps k so that every number reaches a smaller number within at most k applications of the function $C(\cdot)$.

Question 3 is finally something we can resolve. Prove that the answer to Question 3 is "no."

(*Hint:* For an arbitrary positive integer t, consider the number $2^t - 1$. Work out a pattern for what happens when you apply the function $C(\cdot)$ to it repeatedly.)

We've talked about the distinction between "good" and "bad" algorithms for problems involving large integers. 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, or n to some power, than like the number of digits in n.

Thus, we saw that the elementary-school algorithms for addition and multiplication are good algorithms. Less obviously, Euclid's algorithm for greatest common divisors is a good algorithm, as is the recursive algorithm for computing $a^x \bmod n$ that we analyzed in class. 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 ...

---

(6) 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. (Although this is not critical for answering the question, you can assume for simplicity that each public key is publically labeled with the ID number, from 1 through 100, of the agent that possesses it.)

(**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 (or some small power of the number of agents) in its running time, since the number of agents is simply the (small) constant 100.)

(7) 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.