Massachusetts Institute of Technology
6.857: Network and Computer Security
Professor Ronald L. Rivest

Handout 12
October 7, 2003

Problem Set 3 Solutions

Problem 3-1. Prime Triangles.

TA Notes: It is a little-known fact (to us, anyway) that the Java 1.4.1 class library contains a bug in part of its primality tester. Specifically, the Lucas-Lehmer test is implemented incorrectly, and sometimes reports that certain large numbers are composite when they are actually prime[1]. This slowed down our own triangle-finding algorithm for large values of k, and also caused some false alarms during grading. Fortunately, the bug is fixed in version 1.4.2.

Most good solutions (and our own, as well) used the following general framework: pick a value a (it helps if a is prime, so that it's relatively prime to all b and c), then find many values of b that form a "prime pair" with a (i.e., both $a \circ b$ and $b \circ a$ are prime). For each newly-found value of b, see if it forms a prime pair with any previously-found value c; if so, return (a, b, c) as a prime triangle.

For a fast implementation of the above algorithm, there are several optimizations that can be used before employing full Miller-Rabin tests: the Sieve of Eratosthenes, small prime divisor tests, and greatest common divisor tests.

While this strategy works very well, the search time for k-digit prime triangles seems to vary wildly, even for fixed k. This is probably due to the fact that some a values can work in many triangles, while others cannot belong to any at all ($a = 73$ is one example for $k = 2$). The running time, then, dramatically depends upon how lucky we are at choosing a good initial value of a.

It also seems that code written in C (using the GMP toolkit, for example) still has significant performance advantages over code written in Java. Groups that wrote their programs in C typically were able to find triangles for larger values of k than groups who used Java, using comparable algorithms.

For grading, we reserved the top scores (9s and 10s) for groups who described good algorithms and found triangles for $k = 40$ or higher. A decent algorithm and a value of $k = 20$ or higher typically earned 7 to 8 points.

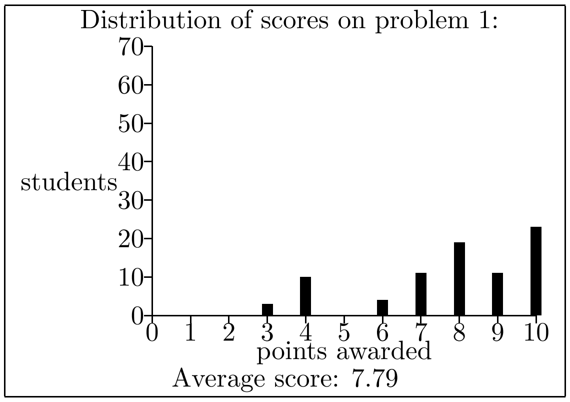
Using an algorithm similar to the one below (but written in C), the group of Alexandros Kyriakides, Saad Shakshir, and Ioannis Tsoukalidis found a prime triangle for $k = 140$.

The following is our (heavily-commented) implementation of a prime triangle-finder, in Java. With it, we were able to find a prime triangle for $k = 100$ in about 30 minutes on an Athlon 1.4 GHz machine running Debian Linux. On average and on the same hardware, we are able to find a prime triangle for $k = 140$ in about 6 hours. The code can be downloaded from the 6.857 website.

```
/**
   A reasonably efficient prime triangle finder, written for Problem 3-1
   of 6.857, Fall 2003.

   @author Chris Peikert
*/

import java.math.*;
import java.util.*;

public class PrimeTriangle {

    // a big enough window, in which we hope to find $b$ and $c$.  A
    // window of 5000000 is sufficient for $k$=70; for larger values
    // of $k$ a wider window is probably desired (it does not affect
    // performance much).
    public static final int SIEVE_SIZE = 20000000;

    // the certainty used by isProbablePrime, though for large enough
    // $k$ the method will override this with a smaller value
    public static final int CERTAINTY = 30;
```

[1]See http://developer.java.sun.com/developer/bugParade/bugs/4624738.html for details.

```

    // an array of the first several prime numbers, used in many places
    public static final int[] prime =
    {
        2,          3,          5,          7,
        11,         13,         17,         19,         23,
        29,         31,         37,         41,         43,
        47,         53,         59,         61,         67,
        71,         73,         79,         83,         89,
        97,         101,         103,         107,         109,
        113,         127,         131,         137,         139,
        149,         151,         157,         163,         167,
        173,         179,         181,         191,         193,
        197,         199,         211,         223,         227,
        229,         233,         239,         241,         251,
        257,         263,         269,         271,         277,
        281,         283,         293,         307,         311,
        313,         317,         331,         337,         347,
        349,         353,         359,         367,         373,
        379,         383,         389,         397,         401,
        409,         419,         421,         431,         433,
        439,         443,         449,         457,         461,
        463,         467,         479,         487,         491,
        499,         503,         509,         521,         523,
        541,         547,         557,         563,         569,
        571,         577,         587,         593,         599,
        601,         607,         613,         617,         619,
        631,         641,         643,         647,         653,
        659,         661,         673,         677,         683,
        691,         701,         709,         719,         727,
        733,         739,         743,         751,         757,
        761,         769,         773,         787,         797,
        809,         811,         821,         823,         827,
        829,         839,         853,         857,         859,
        863,         877,         881,         883,         887,
        907,         911,         919,         929,         937,
        941,         947,         953,         967,         971,
        977,         983,         991,         997,         1009,
    };

    // a corresponding array of the first prime numbers, as BigIntegers
    public static final BigInteger[] primeBig = new BigInteger[prime.length];
    static {
        for(int i = 0; i < primeBig.length; i++) {
            primeBig[i] = BigInteger.valueOf(prime[i]);
        }
    }

    public static void main(String arg[]) {
        if(arg.length != 1) {
            System.out.println("usage: PrimeTriangle <num_digits>");
            return;
        }

        findTriangle(Integer.parseInt(arg[0]));
    }

    /**
       Returns true if the specified number is divisible by a "small"
       prime, and false otherwise.
       @param number the number to test
    */
    static boolean smallPrimeDivides(BigInteger number) {
        for(int i = 0; i < prime.length; i++) {
            if(number.mod(primeBig[i]).equals(BigInteger.ZERO)) {
                return true;
            }
        }
        return false;
    }

    /**
       Returns an array containing the values of the given number, mod
       each prime in the primes array.
    */
    static int[] primeMods(BigInteger number) {
        int[] out = new int[prime.length];
        for(int i = 0; i < out.length; i++) {
            out[i] = number.mod(primeBig[i]).intValue();
        }
        return out;
    }

    /**
       Attempts to find a prime triangle of $k$ digits, prints the
       first found, and returns.  If no prime triangle is found within
       the sieve, prints nothing.
       @param k the desired number of digits
    */
    static void findTriangle(int k) {
        // the value 10; useful everywhere
        BigInteger ten = BigInteger.valueOf(10);
        // the value of the standard multiplier: 10^k
        BigInteger mult = ten.pow(k);
        // the value 10^(k-1), used to ensure that both halves of
        // a number have the right number of digits
        BigInteger other = ten.pow(k - 1);

        // Set $a$ = prime having $k$ digits.  We prefer this because
        // $a, b, c$ must be pairwise relatively prime, so it's
        // easiest just to let $a$ be prime.  The funny business with
        // logs is because the probablePrime method wants the length
        // of $a$ specified in bits.
        int numBits = 1 + (int) Math.round(Math.ceil((k - 1) * Math.log(10) /
                                                     Math.log(2)));
        // We have no need for cryptographic-strength randomness, so
        // just use a weak random number generator to pick $a$.
        BigInteger a = BigInteger.probablePrime(numBits, new Random());
        // The value $a || 00...0$, which we'll use a lot
        BigInteger atop = a.multiply(mult);
        // Now we want to find a _lot_ of primes whose "top halves"
        // are $a$, and whose "bottom halves" are $k$-digit numbers.
        // Therefore we start at $a$ concatenated with 10...0, and
        // work upwards.
        BigInteger baseline = atop.add(other);

        // Next, we use the Sieve of Eratosthenes to quickly rule out
        // most of the numbers in the range [baseline, baseline +
        // SIEVE_SIZE).  The sieve is slightly optimized: only odd
        // numbers are represented in it, so we get double the sieve
        // size from our memory.
        boolean[] sieve = new boolean[SIEVE_SIZE];
        // make it such that: if sieve[i] == true, then (baseline + 2i
        // + 1) is definitely NOT prime.  No need to sieve on the
        // first prime, because it's 2.
        for(int j = 1; j < primeBig.length; j++) {
            // start = smallest index such that (baseline + 2*start +
            // 1) is divisible by prime[j]
            int twoTimesStart = (prime[j] - 1 -
                                 baseline.mod(primeBig[j]).intValue());
            int start;
            if(twoTimesStart % 2 == 0) {
                start = twoTimesStart / 2;
            } else {
                start = (prime[j] + twoTimesStart) / 2;
            }
            // sieve out everything divisible by prime[j]
            for(int i = start; i < sieve.length; i += prime[j]) {
                sieve[i] = true;
            }
        }
        System.out.println("Finished sieving; now looking for triangle");

        // Now find a bunch of numbers for which $a || b$ and $b || a$
        // are prime: in this case, $b$ is called "good."  Every time
        // we find a new good number, check it against all previously
        // found good numbers.  When both concatenations of two good
        // numbers are also prime, we're done!
        Vector good = new Vector();
        // A vector of all the good elements, times 10^k
        Vector goodTop = new Vector();

        // The values of each good number, mod each small prime
        Vector goodMods = new Vector();
        // And time 10^k
        Vector goodTopMods = new Vector();

        for(int i = 0; i < sieve.length; i++) {
            // If still a possibility of primality...
            if(!sieve[i]) {
                // Recall: $b$ corresponds to $other$, plus the twice
                // the sieve offset, plus 1
                BigInteger b = other.add(BigInteger.valueOf(2*i + 1));
                // Construct $b || 00...0$, because it will be used
                // many times.
                BigInteger btop = b.multiply(mult);

                // Test if $a || b$ and $b || a$ are prime.  We do the
                // second test first (and do a fast prime divisibility
                // test before that), because it's more likely to
                // return false and short-circuit (recall the first
                // number, $a || b$, already made it through the
                // sieve).
                BigInteger ba = btop.add(a);
                if(!smallPrimeDivides(ba) &&
                   ba.isProbablePrime(CERTAINTY) &&
                   atop.add(b).isProbablePrime(CERTAINTY)) {

                    // Give us something to look at
                    System.out.print(".");

                    // $b$ is good; add it for future checks
                    good.addElement(b);
                    goodTop.addElement(btop);

                    // Compute $b$ and $b || 00...0$ mod every small
                    // prime, and save the values
                    int[] bmod = primeMods(b);
                    int[] btopmod = primeMods(btop);
                    goodMods.addElement(bmod);
                    goodTopMods.addElement(btopmod);

                    // Check if $b$ works with any other $c$ in the
                    // good vector, i.e. if $b || c$ and $c || b$ are
                    // both prime.
                    for(int j = 0; j < good.size() - 1; j++) {
                        BigInteger c = (BigInteger) good.elementAt(j);
                        BigInteger ctop = (BigInteger) goodTop.elementAt(j);

                        int[] cmod = (int[]) goodMods.elementAt(j);
                        int[] ctopmod = (int[]) goodTopMods.elementAt(j);

                        // Construct $b || c$ and $c || b$
                        BigInteger bc = btop.add(c);
                        BigInteger cb = ctop.add(b);

                        // First, do a lot of cheap tests to see if
                        // one of the values is non-prime.  Test if $b
                        // || c$ or $c || b$ are divisible by a small
                        // prime, with short-circuiting
                        boolean divisible = false;
                        for(int m = 0; m < prime.length; m++) {
                            if((btopmod[m] + cmod[m]) % prime[m] == 0 ||
                               (ctopmod[m] + bmod[m]) % prime[m] == 0) {
                                divisible = true;
                                break;
                            }
                        }
                        if(divisible) {
                            continue;
                        }
                        // if $gcd(b,c)$ != 1, then neither of $b ||
                        // c$, $c || b$ is prime.
                        if(!b.gcd(c).equals(BigInteger.ONE)) {
                            continue;
                        }
                        // Only do expensive isProbablePrime tests if
                        // we've passed the cheap tests
                        if(bc.isProbablePrime(CERTAINTY) &&
                           cb.isProbablePrime(CERTAINTY)) {
                            System.out.println("\ntriangle = (" + a +
                                             "," + b + "," + c + ")");
                            return;
                        }
                    }

                }
            }
        }
    }
}
```



Problem 3-2. Block Ciphers.

The Block Cipher problem asked you to design a hardware disk encryption device that would fit between a computer and the disk. The device is to automatically encrypt data written to the disk and decrypt information read back from the disk.

This approach to hardware disk encryption has been used at various times by devices sold on the market. It is also the idea behind several software-based disked encryption systems that operate at the device driver layer. Some of the advantages of block-based encryption schemes include:

- If the disk is removed from the encryption device, its contents cannot be accessed. This is useful if the disk needs to be discarded or serviced.
- Operating block-by-block, the encryption scheme is completely transparent to the operating system. This means it will work with any operating system. File-based encryption schemes are operating system-dependent.
- Typically, these schemes are easier to debug than file-based systems, because they have a simpler interface to the operating system.
- Also, these systems have a very simple user interface. You can store the user's key on a token such as a USB dongle. When the dongle is inserted, the data on the disk can be read and decrypted; when the dongle is not present, data on the disk is protected.

In reading student answers, we found several confusions regarding encryption modes of operation. Some of the more common errors are summarized below:

- **Confusion over disk blocks and cipher blocks.** Hard disks use a block size of 512 bytes. The AES encryption algorithm uses a block size of 64 bits, or 8 bytes. So there are 64 AES blocks in each disk block.

  Because AES blocks are smaller than disk blocks, the disk encryption system needs to use the AES cipher in one of the recognized modes of operation. Most groups suggested using Cipher Block Chaining

(CBC) mode, as CBC mode is more secure than ECB mode. The problem with ECB mode is that the same block of data encrypted twice encrypts the same way; this makes it possible to pick up patterns in the ciphertext that might be otherwise hidden with a better encryption mode. A few groups realized that a problem with CBC mode is that you cannot start encrypting or decrypted in the middle of the cipher-stream — you can only start at the beginning. The correct solution here is to treat each 512-byte disk block as an independent cipher-stream.

- **Confusion over strength of AES.** Most groups decided to use AES as their encryption algorithm. Those that didn't were generally wrong, and their justification of their choice was in error. One group thought that AES was too slow, and decided to use 3DES instead. (3DES is slower than AES.) One group said that AES was unproven and 3DES was safer (in fact, AES had a termendous amount of analysis in terms of people-years, because of the nature of the competition under which it was chosen.) One group said that AES had been proven to be secure. (Alas, not true: we don't have the math to make such proofs at this point in time.) One group said "there is no danger of the encryption scheme itself being cracked by a malicious attacker." Nope: AES could be cracked tomorrow. But so could 3DES.
- **Confusion over IV.** When a block cipher is run in CBC mode, the ciphertext of each block depends on the key, the plaintext of the current block, and the ciphertext of the previous block. For the first block there is no previous block, so an *initialization vector* is specified.

  Many groups in the class thought that the IV needed to be kept secret, or it needed to be a random number. There are no such restrictions! The two important things in choosing an IV is that it should be different every time, and you need to remember the IV somewhere so that you can decrypt the ciphertext.

  A common error on this problem set was using an IV generated from a random process within the disk drive. You're welcome to do that, but if you do that, you'd better remember the IV somewhere. Otherwise you won't be able to decrypt the data! It's much more convenient and simple to just use each disk block number as the IV for that block.
- **Data Encryption Key vs. User Key.** Clearly, the use of any encryption system requires a key. Some groups suggested that the disk encryption key be created in the system when it is created and be unchangeable. This is actually a reasonable approach if you are primarily concerned about the sanitization of discarded hard drives. A more flexible approach ties the key to a user passphrase. The problem with this approach, though, is that you don't want to have to encrypt and re-encrypt the entire disk drive if the user wishes to change her passphrase (or key). The solution is to have a Data Encryption Key (DEK) that is stored inside the device, and to have this DEK itself encrypted with the User Key. This User Key can be kept on a dongle or it can be derived from a passphrase that the user provides. To change the user passphrase, simply decrypt the DEK with the old passphrase and encrypt it with the new one.
- **Confusion over performance.** Many groups incorrectly stated that hardware-based systems are faster (or generally faster) than software-based systems. In fact, there is really no reason why hardware systems should be slower or faster than software systems: speed is determined by the device clock rate and by how much real work (in this case, how much encryption or decryption) can be done during each clock cycle.

Other routine errors that were made in the write-ups:

- Even though AES is more modern than DES and uses a larger key, it is actually faster in software (if implemented properly) and easier to implement in hardware.

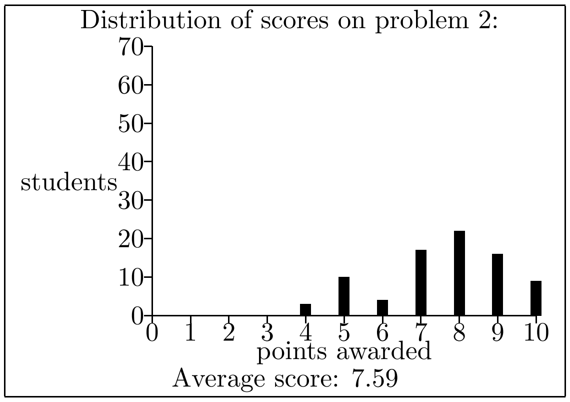
Grading.

- 2 points were awarded for choosing AES and properly justifying that choice.

- 2 points were awarded for choosing CBC, OCB, or OFB modes of operation (and for making a good argument as to why that mode should be chosen).
- 2 points were awarded for choosing an IV that could be recovered. (Groups that suggested setting the IV from a random variable such as the average disk drive speed lost credit unless they specified that each IV be recorded somewhere on the disk.) Note: OCB mode calls the IV a *nonce*, but they're basically the same thing.
- 2 points were awarded for having a separate Data Encryption Key from a user passphrase to allow easy key changing. Some other equally clever (and secure) solution would also earn full credit.[2]
- 2 points were awarded for general writing style.

Points were removed at the TAs discretion for technological proposals that would not work. And, as always...

- 2 points were subtracted for using the first person singular pronoun.



Problem 3-3. MAC Attack.

The first key to solving the MAC Attack problem revolved around the realization that the same block encrypted with ECB will always produce the same ciphertext, no matter what encryptions have come before. Everybody figured that out.

The second key to solving this problem revolved on the realization that the black box will always add a 1 followed by between 0 and 127 0s, depending on the number of bits in the final 128-bit block of the message. This is what is meant by "she unconditionally appends a 1 bit . . . "

Groups who successfully solved this problem employed one of several strategies. The most common strategy involved computing the multiplicative inverse of the MAC of a message that was less than 128 bits in length using this algorithm described below.

Let's start with a simple message: $M_0 = 1^{127}$.

Remember, the black-box pads this message to 1^{128} before encrypting. Let's call that M_0'.

We send this to the black box and get back MAC_0:

$$\begin{aligned}\text{MAC}(M_0) &= \sum_{i=0}^{0} \text{AES}_k(M_0')a^0 \pmod p \\ &= \text{AES}_k(M_0') \pmod p\end{aligned}$$

[2]One team suggested splitting the 128-bit key into a 64-bit piece that is stored with the device and another 64-bit piece that is stored on the dongle — which effectively lowers the key to 64-bits. A better approach would have been to split the 128-bit key into two pieces that need to be XORed together.

Now we compute the MAC on the second message M_1 where $M_1 = 1^{255} = M_0 \circ 1 \circ M_0$. This gets padded to $M_1' = 1^{256} = M_0' \circ M_0'$, and the MAC is calculated as:

$$\text{MAC}(M_1) = \text{AES}_k(M_0') + a \cdot \text{AES}_k(M_0') \pmod p$$

Subtracting $\text{MAC}(M_0)$ from $\text{MAC}(M_1)$ we get:

$$\begin{aligned}\text{MAC}(M_1) - \text{MAC}(M_0) &= \text{AES}_k(M_0') + a \cdot \text{AES}_k(M_0') - \text{AES}_k(M_0') \pmod p \\ &= a \cdot \text{AES}_k(M_0') \pmod p\end{aligned}$$

And recall, $\text{AES}_k(M_0')$ is really $\text{MAC}(M_0)$, so we have:

$$\begin{aligned}\text{MAC}(M_1) - \text{MAC}(M_0) &= a \cdot \text{MAC}(M_0) \pmod p \\ \text{MAC}(M_1) &= (a+1) \cdot \text{MAC}(M_0) \pmod p\end{aligned}$$

Notice that we know $\text{MAC}(M_0)$, $\text{MAC}(M_1)$ and p. At this point, several groups said that it is trivial to calculate a. In fact, it isn't "trivial" — you need to know to calculate the multiplicative inverse of $\text{MAC}(M_0) \pmod p$ using Fermat's little theorem or Euclid's Algorithm, as shown in class. Multiply each side by $\text{MAC}(M_0)^{-1}$ and subtract one and you have:

$$\text{MAC}(M_1)\text{MAC}(M_0)^{-1} - 1 = a \pmod p$$

Now that we know a, we can calculate the MAC of the particular message $M_0 \circ M_1 = 1^{128+255}$ (having used only *two* black-box queries!). In fact, we can forge the MAC of an arbitrary message B, using a few more queries. First, pad B in the standard way, so that its length is a exact multiple of 128 bits. Next, break the message up into 128 bit blocks, which we call B_i. For each of these blocks, append 1^{127}, and call the resulting messages B_i'.

Send each of these (appended) blocks to the black-box to learn $\text{MAC}(B_i')$. Notice that:

$$\text{MAC}(B_i') = \text{AES}_k(B_i) + a \cdot \text{MAC}(M_0) \pmod p$$

By subtracting $a \cdot \text{MAC}(M_0)$ from each side, we get:

$$\text{MAC}(B_i') - a \cdot \text{MAC}(M_0) = \text{AES}_k(B_i) \pmod p$$

Using a and the values $\text{AES}_k(B_i)$, the MAC of the desired message can now be calculated using the original MAC definition.

...

It turns out that there was another clever way to solve this problem (without taking modular inverses) that was discovered by several groups. **The following solution was submitted by Andrew Tsai, Dian Chen and Siddique Khan:**

Let: A, B, and C be any 127-bit messages. Let + denote concatenation.

Construct these messages:

$$\begin{aligned}M_1 &= A \\ M_2 &= A + 1 + B \\ M_3 &= C\end{aligned}$$

The black box pads M_1 and M_2 with a 1 so that the messages are now $A+1$ and $A+1+B+1$, respectively. The black box returns:

$$\begin{aligned} \mathrm{MAC}(A+1) &= c_0 \pmod p \\ \mathrm{MAC}(A+1+B+1) &= c_0 + c_1 a \pmod p \end{aligned}$$

We now compute:

$$\mathrm{MAC}(A+1+B+1) - \mathrm{MAC}(A+1) = c_1 a \pmod p$$

We then pass M_3, which becomes $C+1$ when padded, to the black-box and record:

$$\mathrm{MAC}(C+1) = c_0'$$

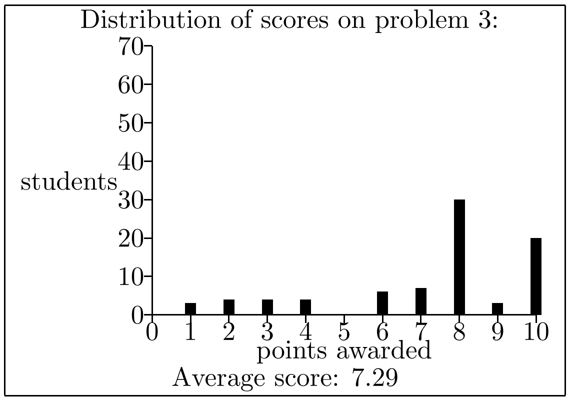
Finally, we define M_4 to be:

$$M_4 = C + 1 + B$$

The black box appends a 1, so M_4 becomes $C+1+B+1$ which is what the MAC would be performed on. We can thus forge the MAC that a black box would spit out on an input of M_4 since we have $\mathrm{MAC}(C+1)$ and $\mathrm{MAC}(A+1+B+1) - \mathrm{MAC}(A+1) \pmod p$:

$$\begin{aligned} \mathrm{MAC}(C+1) + \mathrm{MAC}(A+1+B+1) - \mathrm{MAC}(A+1) &= c_0' + c_1 a \pmod p \\ &= \mathrm{MAC}(C+1+B+1) \end{aligned}$$

Note that A, B and C can actually be any messages of equal length such that their length mod 128 is 127 bits.


Distribution of scores on problem 3:
students
points awarded
Average score: 7.29