CS441, Spring 1995 **Assignment 1**
Primaltiy Testing **Due: 29 April (11:59pm)**

Implement part of a program that tests large integers for primality.
Large numbers can be difficult to break into prime factors. Modern cryptographic methods depend on this fact. In the RSA public-key cryptography system, someone who wishes to receive messages publishes a number K that's the product of two large (e.g., 200-digit) prime numbers P and Q. Anyone knowing K can encode a message, but only the person who knows P and Q can decode the message.
How can the receiver find some large primes P and Q? The simplest way is to choose a random 200-digit number, and test whether it is prime. But how does one test whether a number N is prime? We can try dividing by each prime number up to $\sqrt{N}$. But there are approximately 4×10^{97} primes less than $\sqrt{10^{200}}$, so at the rate of one per microsecond it would take 10^{74} times the age of the universe to test N in this way. Fortunately, it is possible to learn that a number is composite (not prime) without even learning its factors. This can be done in a more reasonable length of time. We'll use the following mathematical fact about prime numbers; for a prime number p and a nonzero number a,

$$a^{(p-1)} \bmod p \;=\; 1$$

But for a typical composite c, $a^{c-1} \bmod c \;\neq\; 1$ for at least half the a's.
We can test a number n for primality by choosing a random a, raising it to the $(n-1)^{\text{th}}$ power modulo n, and seeing if we get 1. If we get something other than 1, then we know n is composite. If for k random a's we get 1, then the chance that n isn't prime is about 2^{-k}. So if we do a hundred tries and always get 1, then n is almost certain to be prime.
So, to make a primality testing program, we read in a number n, choose forty or a hundred random a's and compute $a^{n-1} \bmod n$. If we always get 1, we report "probably prime." Otherwise, we report "definitely composite."

Big numbers

Now the only problem is that an `int` on our computer can hold only a 32-bit (10 decimal digit) number, not a 200-digit number. So how do we do exponentiation and modulus on these big numbers?
Let's represent a number as an array of digits in the base b. You are used to base 10, of course; but we will use base $b = 65536$ since that's convenient on the computer. An `unsigned short` can hold numbers in the range 0..65535, so a 200-digit decimal number is just an array of about 80 `unsigned short`s. Or, with base 4294967296, an `unsigned long` can hold each digit.
These numbers can be added, subtracted, and multiplied by the same rules you learned in grade school (before they let you use pocket calculators). Remember carries and borrows? Just start at the low-order digit and work your way up.
To do the `mod` operation, we need to do long division and keep the remainder. Here are some mathematical rules regarding division:

$$a < b \quad\rightarrow\quad a/b \;=\; 0 \text{ rem } a$$

$$a/2b = q \text{ rem } r, \quad r < b \quad\rightarrow\quad a/b \;=\; 2q \text{ rem } r$$

$$a/2b = q \text{ rem } r, \quad r \geq b \quad\rightarrow\quad a/b \;=\; 2q + 1 \text{ rem } r - b$$

We also have the problem of exponentiating numbers. To raise a to the power k, all we have to do is multiply a by itself k times. The problem is that if $k = n - 1$, and n is a 200-digit number, this will take much longer than the age of the universe. But we can use the following identities:

$$k \text{ even} \quad\rightarrow\quad a^k = (a^{k/2})^2$$

$$k \text{ odd} \quad\rightarrow\quad a^k = a \cdot a^{k-1}$$

But there's still a problem here. The number a is likely to be about 200 decimal digits, and if we take it to the 10^{200} power then we get a number with more digits than atoms in the universe; this won't fit onto

our computer. But we don't really have to compute a^{n-1}, we have to compute $a^{n-1} \bmod n$; this is a much smaller number, of about 200 decimal digits. We can use the following mathematical identity while doing the exponentiation:

$$(ab) \bmod n \quad = \quad (a \bmod n)(b \bmod n) \bmod n$$

Thus, we can keep all the intermediate results down to 400 digits (or 200, if we're clever *during* the multiply; but don't worry about this).

Input/Output

The program must read a decimal number (an ASCII character string) from the standard input. To read a decimal number, just multiply values in the range 0...9 by powers of ten and sum them up.
For debugging it might be useful to print numbers in decimal. To convert a number into decimal format, you must start by calculating the last digit; just divide by ten and take the remainder.

Program specification

The program "prime" must read a sequence of arbitrary-precision decimal numbers from the standard input. Each line will be a non-empty sequence of decimal digits, terminated by a newline. For each line, the program must report "probably prime" or "composite" on the standard output, with each answer terminated by a newline. On ill-formed input the program must print an error message and terminate cleanly.

Your assignment

In the precept section, discuss how to modularize this program and define the interfaces between modules. Discuss the mathematical algorithms for each of the arithmetic operations. Then your programming assignment is to implement two of the modules, as chosen by you and the preceptor. Each module will probably be about 35 lines of code, more or less.
You must rigorously test your modules, since bugs in your modules will cause the whole program to fail. Methods for testing will be discussed in precept. Make sure to put in assertions. Part of what you turn in should be a description of how you tested, or otherwise ensured the correctness, of your modules.
It is not cheating to test together with someone who is working on a different module. You must ensure, of course, that you have been responsible for designing and implementing your own module. But testing might be much easier if you can hook more than one module together.

Grading

Your grade for this program will be based on your participation in precept, on the test strategy you have designed, on the correctness of your program, and on the elegance and readability of your program and documentation.

Submission

Submit your two modules and a description of how you tested them using the command:

```
/u/cs217/bin/submit 7 ....
```

Notes

It couldn't hurt for you to understand the algorithms used by the modules you don't have to implement.
You will have to do something about freeing the bignums that are results of intermediate calculations, or else they will eat up all of memory.