

CS 1160-01 Introduction to Computer Science and Programming Methods
Programming Assignment 3
Due, Monday, February 7, 2005

Hunting for Primes (Apprentice Grade)

This program will accept as input a *positive integer* n and will return the first *prime* number greater than or equal to n . It will use a simplified version of the Rabin-Miller primality test. The Rabin-Miller test actually finds *probable primes*—numbers which have a high probability of being prime. (These are sometimes jokingly called *industrial grade primes*. For example, the federal Digital Signature Standard (FIPS-186—FIPS publications can be found on-line by searching from www.fedworld.gov) recommends use of the Rabin-Miller test for choosing primes.)

The general idea of the test is that in order to test a number n for primality, one chooses a *random* integer a (sometimes called a *witness*) between 2 and n and then performs a test calculation involving a and n . If n is *not* prime, the underlying mathematics says that the probability is $\frac{3}{4}$ that the test will *fail*. This means that if n *passes* the test for a number of *independently chosen* witnesses, it is highly likely to be a prime. (FIPS-186 suggests that 80 such tests should be “good enough”, even for 300-digit primes.) Your program will only deal with fairly small integers, and may simply use a *sequence* of witnesses instead of random choices.

What do we need?

1. You’ll need a function with prototype, say, `int fast_exp(int base, int expon, int modulus)`, which for `base` b and `expon` e and `modulus` m will compute the value $b^e \bmod m$. (Note: “mod” is pronounced “%” in C++.) To make this *fast* or *efficient*, you should compute using successive squaring, i.e.,

- i. Initialize a *temp* variable, say, `tmp`, to 1. Then while `expon` is positive, repeat a loop that does three things:
- ii. If `expon` is *odd*, replace `tmp` by `(tmp*base)% modulus`,
- iii. Replace `base` by `(base*base) % modulus`, and
- iv. Replace `expon` by `expon/2`.

NOTE: To avoid the dreaded *arithmetic overflow*, you should declare `tmp` and a local variable to serve as `base` as type “long long int”, i.e., as 64 bit integers. This type is supported by most modern compilers. If it is not supported on yours, then this function will only work when `base` $< 2^{15}$ or so.

2. You’ll need a function to test a candidate n using a witness a ; this function can return a `bool` value. The test goes like this:

- i. Rewrite n as $n - 1 = m * 2^k$, where m is *odd*. That is, 2^k is the highest power of 2 that divides $n - 1$ exactly.
- ii. For some `temp`, say, `ttmp`, set `ttmp` equal to `fast_exp(a, m, n)`. If this is either 1 or $n - 1$, the test is **passed**, and the test is finished.
- iii. If not, then successively replace `ttmp` by `fast_exp(ttmp, 2, n)`. Do this $k - 1$ times. If any of the values so computed are equal to $n - 1$, then the test is **passed**, and the test is finished.
- iv. Otherwise, the test is **failed**, and n is *definitely not* a prime.

3. You will need a main program—which, itself, you may want to break into more than one function—which will start with a *user supplied* integer n and will test n , $n + 1$, etc. until one turns up as a *probable prime*. [NOTE: *Even* integers can be rejected immediately.]

When testing a given n , make sure it **passes** the test for at least 80 witnesses. Instead of choosing random witnesses a , it will be sufficient to use a simple loop that uses the numbers $2, 3, \dots, 81$ as witnesses. (This assumes, of course, that n is larger than 81.)

Once your program finds a probable prime, it should report it to the user.

What to turn in

As usual, you should turn in your *source code*, which should be in good style and well commented. You should also turn in the *results* of enough runs of your program to convince one that it works correctly.