Primality Testing

Goals of Assignment

- Writing software as part of a large team
- Living and breathing what COS 217 is about
  - Abstraction, separation of interfaces and implementations, modularity
- Also, more on …
  - Advanced C programming
  - Creating and using ADTs
  - GNU/UNIX programming tools
- Bonus: Learn a little about implementing security

Why Test for Primality of Numbers

- Modern cryptographic methods depend on a key fact
  - Large integers can be difficult to break into prime factors
- RSA public-key cryptography system
  - One who wants to receive messages publishes an integer k, that is the product of two large (e.g., 200-digit) prime integers p and q
  - Anyone who knows k can encode a message
  - But only the person who knows p and q can decode the message
  - Finding out p and q from k is hard, for very large k

How to find large primes p and q?

- Simplest way: choose a random 200-digit integer, and test whether it is prime
- How to test whether an integer n is prime?
- Could try dividing by each prime integer up to sqrt(n)
- You'd be waiting a while …
  - 200-digit integer n is of size up to 10^{200}
  - There are approximately $4 \cdot 10^{97}$ primes less than sqrt(10^{200})
  - At the rate of one per microsecond, this method would take you 10^{74} times the age of the universe to test n

Fortunately …

- Can learn that an integer is composite (i.e. not prime) without even learning its factors, and in reasonable time
- Mathematical facts
  - For a prime integer p and an integer a in the range $1 \le a < p$:
    $$a^{p-1} \bmod p = 1$$
  - But for a typical composite integer c:
    $$a^{c-1} \bmod c \ne 1 \text{ for at least half the a's.}$$
- So, to test an integer n for primality:
  - Choose a random a (in $1 \le a < n$)
  - Raise it to the (n-1)st power modulo n, and see if you get 1
  - If not, n is composite. If so, it could be prime or composite
  - Try k times. If for k random a's you get 1, then the chance that n isn't prime is about 2^{-k}
  - If say 40 tries result in 1 each time, n is almost certain to be prime

Primality Testing Program

- Read in an integer n
- For each of 40 randomly chosen a's (s.t. $1 \le a < n$), compute $a^{n-1} \bmod n$
- If result is 1 all 40 times, report: "*n is probably prime*"
- If any one of 40 tries yields something other than one, report: "*n is composite (i.e. not prime)*"

a=3 n=7 $a^{n-1} \bmod n = 3^{7-1} \bmod 7 = 1$

a=5 n=3 $a^{n-1} \bmod n = 5^{7-1} \bmod 7 = 1$

7 is probably prime

a=8 n=9 $a^{n-1} \bmod n = 8^{9-1} \bmod 9 = 1$

a=5 n=9 $a^{n-1} \bmod n = 5^{9-1} \bmod 9 = 7$

9 is composite (for sure)

- **Looks good, but …**

BigInts …

- We're talking about doing exponentiation and modulus etc on 200-digit (decimal digit) integers
- The hats computers can only store 32-bit integers (10 decimal digits)
- Solution: represent a big integer as an array of digits in the base b
- What is b?
  - Unsigned int can hold integers in the range 0..4294967295
  - So use base b = 4294967296, so that each unsigned int on hats represents a "digit" in base b
  - A 200-digit decimal integer is just an array of about 21 unsigned ints, so 21 "digits" in this base b representation
- Let's look at how we do arithmetic with BigInts

BigInt Addition and Subtraction

- Use same rules that you learned in grade school for decimal digits
- Carries, borrows …
- Just start at the low-order digit and work your way up

```
  1 1 1   <- carries
314159265
+  928541
---------
315087806
```

```
  1  1    <- borrows
314159265
-  928541
---------
313230724
```

Detecting carry and borrow

When using an unsigned long to implement a "1-digit" add (or subtract),

how do you detect carry (or borrow)?

```
unsigned long x = 3018591856 +
                  2847567187;
printf("=%d",x); =1571191747
```

(true answer is **5866159043**, but that's not what you get!)

by the way, **1571191747 = 5866159043 mod** $\mathbf{2^{32}}$

BigInt Multiplication

- To multiply, use a recursive approach based on these mathematical rules:
  - $b = 0 \Rightarrow a \cdot b = 0$
  - b even $\Rightarrow a \cdot b = (a \cdot b/2) \cdot 2$
  - b odd $\Rightarrow a \cdot b = ((a \cdot b/2) \cdot 2) + a$

where "/" is truncating integer division. (Recall that multiplying and dividing by 2 on a digital computer is easy)

$$
\begin{aligned}
&93 \cdot 13 = \\
&(93 \cdot 6) \cdot 2 + 93 = \\
&((93 \cdot 3) \cdot 2) \cdot 2 + 93 = \\
&(((93 \cdot 1) \cdot 2 + 93) \cdot 2) \cdot 2 + 93 = \\
&(((93 \cdot 0 + 93) \cdot 2 + 93) \cdot 2) \cdot 2 + 93 = \\
&(((0 + 93) \cdot 2 + 93) \cdot 2) \cdot 2 + 93 = \\
&1209
\end{aligned}
$$

BigInt Division and Modulus

- Recursive approach based on these mathematical rules:

  a < b ⇒ a/b = 0 rem a

  a/(2·b) = q rem r ⇒ r<b ⇒ a/b = (2·q) rem r

  a/(2·b) = q rem r ⇒ r≥b ⇒ a/b = (2·q) + 1 rem (r-b)

1200 / 13	=	92 rem 4
1200 / 26	=	46 rem 4
1200 / 52	=	23 rem (56-52) = 23 rem 4
1200 / 104	=	11 rem (160-104) = 11 rem 56
1200 / 208	=	5 rem (368-208) = 5 rem 160
1200 / 416	=	2 rem 368
1200 / 832	=	1 rem (1200-832) = 1 rem 368
1200 /1664	=	0 rem 1200

BigInt Exponentiation

- For a^k, need to multiply a by itself k times
  - If k = n-1, will take much longer than age of universe
- But we can use the following identities:
  - k even $\Rightarrow a^k = (a^{k/2})^2$
  - k odd $\Rightarrow a^k = a \cdot a^{k-1}$

$$
\begin{aligned}
&6^{10} = \\
&(6^5)^2 = \\
&(6 \cdot (6^2)^2)^2 = \\
&(6 \cdot (36)^2)^2 = \\
&(7776)^2 = \\
&60466176
\end{aligned}
$$

BigInt Exponentiation (contd.)

- Another problem: sizes of numbers
  - a and k are both 200-digit numbers
  - taking a 200 digit number to the 10^{200} power gives an integer with more digits than atoms in the universe
  - won't fit on little ol' hats ....
- But, don't really have to compute a^{n-1}
  - only a^{n-1} mod n, which is smaller integer, only 200 decimal digits
- Can use this mathematical identity for exponentiation:
  - $(a \cdot b) \bmod n = ((a \bmod n) \cdot (b \bmod n)) \bmod n$
- Thus, can keep all intermediate results down to 400 digits
  - or 200, if you're clever *during* the multiply, but don't worry about that
- Read assignment for all the rest …

Sanity check

- Will all this stuff run fast enough?

Answer: do a quick big-Oh analysis

Dividing into modules

- You’ll need the basic operations for primality testing (add, subtract, divide-with-remainder, exponentiation) as well as conversion to/from decimal, and perhaps some debugging functionality
- Pay particular attention to: which algorithms need to see the ***representation*** of a bigint, and which do not
- You may have to decide on memory-management (malloc/free) protocols …
- The “dc” utility on Unix can be really useful in checking answers during debugging; do “man dc” at the shell prompt, or google “dc man page”