Math 55 - Spring 2004 - Lecture notes # 9 - Feb 19 (Thursday)

Finish through end of Section 2.6 (not 2.7) if not yet done
Start reading Sections 3.1 - 3.4

Homework, due Feb 25

(1) Let a = a4a3a2a1a0 be a 5-bit 2s complement integer;
each ai is 0 or 1. Similarly let b = b4b3b2b1b0, and
let s = a+b = s4s3s2s1s0 be their sum in
2s complement arithmetic.
Interpreting a 0 bit as False and a 1 bit as True, write down
logical formulas for s4,s3,s2,s1,s0 using and, or, not, xor,
with the inputs a4,a3,a2,a1,a0, b4,b3,b2,b1,b0.
Hint: introduce new logical variables (bits) c4,c3,c2,c1,c0
where ci is the carry into the i-th bit from the i-1-st bit.
Your logical formulas for si and c(i+1) in terms of ai, bi
and ci should look the same for all i
(you can let c0 = 0 = False so
that all the formulas look the same).
In particular, your formulas should express the facts that
c(i+1)=1 if the sum ai + bi + ci is at least 2, and
si = 1 if ai + bi + ci is 1 or 3 (i.e. odd).
Computers implement these formulas in hardware to perform
2s complement addition.

(2) Explain how to use nearly the same logical formulas as above
to compute the difference d = a-b

(3) The purpose of this question is to illustrate that
there are a lot of primes.

(a) Let n and d be integers, and x = n*10^d
Use the prime number theorem to evaluate the
limit as d -> infinity of pi(x + 10^d) / pi(x)
where pi(x) is the number of primes less than or equal to x.
(in other words, n is fixed and d is growing)

(b) Use part(a) to show that given any arbitrary string of
decimal digits (representing the integer n),
then for all sufficiently large M, there is always a prime
p such that

1. p has an M-digit decimal expansion, and
2. p's decimal expansion starts with the given string
(representing n).

2.4-46

2.5-18,36,38,40

```
     2.6-20,24

Goals for today: Recall Euclidean algorithm for the gcd
                 Use it to solve a "congruence equation"
                      a*x=b mod m for x
                      how to do division in modular arithmetic
                 Use it to solve a system of congruence equations:
                      Chinese Remainder Theorem
                 Apply it to cyptography

Recall property of Euclidean algorithm for gcd: given a and m,
it computes
    1) d = gcd(a,m)
    2) integers s and t such that a*s+m*t=d

How to solve a*x == 1 mod m for x:
(analogy of reciprocal of a in modular arithmetic)
Theorem: a*x ==1 mod m can be solved for x if and only if gcd(a,m)=1.
       When it can be solved, x is unique mod m, i.e. the only one in
       the range 0 to m-1, and is called "the inverse of a modulo m".
  EX: Solve 2*x==1 mod 5: try x=0,1,2,3,4,
                   getting 2*x=0,2,4,1,3,
      so x=3 is the unique answer (gcd(2,5)=1)
  EX: Solve 2*x==1 mod 4: try x=0,1,2,3,
                   getting 2*x=0,2,0,2,
      so there is no solution (gcd(2,4)=2)
  Proof: If gcd(a,m)=1, we have to show that we can solve for x:
         Use the Euclidean algorithm to find s and t such that
         a*s+m*t=1. Thus a*s = 1-m*t == 1 mod m, so x=s is a solution.

         If gcd(a,m) /= 1, we have to show that no x satisfies
         a*x==1 mod m: Recall that a*x == 1 mod m is equivalent to
         a*x mod m = 1 mod m = 1, and that a*x mod m = a*x+m*t
         for some t.  But if gcd(a,m)=d>1, then d|a and d|m,
         so d|(a*x+m*t) for any integer t, and in particular
         d|(a*x mod m).  Since d does not divide 1, a*x mod m /= 1.

         To show that the solution x is unique mod m when it exists,
         suppose both that a*x1 == 1 mod m and a*x2 == 1 mod m, and
         that 1 <= x1 < m and 1 <= x2 < m; we have to show that x1=x2.
         Now a*x1-a*x2 == 0 mod m, so m|(a*(x1-x2)).
         Since gcd(a,m)=1, a and m have no common factors,
```

```
          and thus m|(x1-x2). Now x1-x2 satisfies two properties:
            1) m|(x1-x2), so x1-x2 is in the set
                {..., -2*m,-m,0,m,2*m,...}
            2) -m < x1-x2 < m, since 1 <= x1 < m and 1 <= x2 < m;
          The only value of x1-x2 satisfying these properties is
          x1-x2=0, or x1=x2 as desired.

   Corollary: a*y == b mod m has a solution y for any b if and only if
       gcd(a,m) == 1  (analogy of dividing b/a in modular arithmetic)
       proof: if gcd(a,m)=1, then the Theorem says we can solve
              a*x == 1 mod m. Multiply through by b to get
              a*(x*b) == b mod m, so we can take y = x*b
              (b times "inverse of a") If gcd(a,m)>1,
              then the Theorem tells us we cannot solve when b=1.

ASK&WAIT: under what conditions on b can we solve a*y == b mod m for y?

   Chinese Remainder Theorem: Let m1, m2,..., mn be pairwise
       relatively prime numbers, ie gcd(mi,mj)=1 for all i and j.
       Let m = m1*m2*...*mn.  Then the n equations
         x == a1 mod m1 , x == a2 mod m2, ... , x == an mod mn
       have a unique solution mod m for any a1, a2,...,an, i.e. there is
       only one solution in the range from 0 to m-1.
   EX: x == 2 mod 3, x == 3 mod 5
           x    x==2 mod 3 ?   x==3 mod 5?
           0
           1
           2        Yes
           3                      Yes
           4
           5        Yes
           6
           7
           8        Yes           Yes  x=8 is unique solution mod 3*5=15
           9
          10
          11        Yes
          12
          13                      Yes
          14        Yes

    Proof: We give an algorithm for computing x, and leave uniqueness to
```

```
      homework. Let Mi = m/mi, for i=1,...,n. Thus
      Mi = product of all mj except for mi, so gcd(Mi,mi)=1, since
      mj and mi have no common factors. By the last theorem, each
      Mi has an inverse yi mod mi, i.e. Mi*yi == 1 mod mi.
      We claim a solution is x = a1*M1*y1 + a2*M2*y2 + ... + an*Mn*yn.
      To confirm this we have to verify that x == ai mod mi for all i:
      x == (a1*M1*y1 + a2*M2*y2 + ... + an*Mn*yn) mod mi
        == ( a1*M1*y1 mod mi + ... + an*Mn*yn mod mi ) mod mi
        == ( ai*Mi*yi mod mi
           + sum_{j /= i} aj*Mj*yj ) mod mi
        == ai mod mi       since Mi*yi == 1 mod mi
           + 0             since mi | Mj when j /= i
        == ai mod mi       as desired

 EX: x == 2 mod 3, x == 3 mod 5 again:
     a1=2, m1=3, a2=3, m2=5, M1=5, M2=3,
     y1=2 since 2*5==1 mod 3   and y2=2 since 2*3==1 mod 5
     x = 2*5*2 + 3*3*2 = 38 == 8 mod 15

Cryptography
   Recall that a message (character string) is converted to a number M
   What happens when a Sender wants to send a secret message to
   a Receiver:
     The Sender takes message M and encrypts it to get the
          encrypted message C = f_enc(M)
     The Sender sends C to the Receiver. Anyone may "intercept"
          C on its way.
     The Receiver decrypts C to get the original message M = f_dec(C).

   For this to work as the Sender and Receiver desire:

     f_enc and f_dec have to be one-to-one, onto functions and be
       inverses of one another, i.e. M = f_dec(f_enc(M)) for all M
     It is easy for the Sender to evaluate f_enc
     It is easy for the Receiver to evaluate f_dec
     It is very hard for anyone other than the Receiver to evaluate
       f_dec.  The harder it is, the better the secrecy.

   Two kinds of cryptography:
     Private key (traditional): need one "Key" for both f_enc and f_dec
       where K=Key is a shared secret between Sender, Receiver
EX: shift:   C = f_enc(M) = M-K mod n, M = f_dec(C) = C+K mod n,
```

```
      easy to break
ASK&WAIT: How?
  EX: xor:  C = f_enc(M) = M xor K (thinking of M, C, K as bit strings
            of the same length)
            M = f_dec(C) = C xor K
ASK&WAIT: Why are f_enc and f_dec inverses?
      hard to break if K used once
  EX: Original Washington/Moscow hotline worked this way
  EX: crypt command in UNIX, uses algorithm from German Enigma machine
      used in World War II, which was broken by Turing

  Secrecy depends on keeping K a secret known only to Sender, Receiver
  so only they can evaluate f_enc and f_dec
  Disadvantage: if 1000 people want to talk to one another in secret,
  need 999*1000 secret keys, so all pairs can talk; too many keys!

  Public key: any Sender can do f_enc, but only one Receiver can do f_dec
  Advantage: for 1000 people to talk in secret, each person has his/her
       own secret f_dec, but can just publish the corresponding f_enc
  EX: RSA (Rivest/Shamir/Adleman)
    Need: 1) large number n that is product of two large primes p*q=n
            large means 200 to 400 decimal digits
          2) integer e that is relatively prime to (p-1)*(q-1)
          3) integer d = inverse of e mod (p-1)*(q-1)
    Everyone knows n and e, but only Receiver  knows d
   Then for message M, C = f_enc(M) = M^e mod n is the encryted message
      For encrypted message C, M = f_dec(C) = C^d mod n is the decrypted
          message
  EX: Try 2537=n=p*q=43*59, e=13, message = STOP = (ST,OP)=(1819,1415)
      using position of letters in alphabet.  Then encrypted message
      = ( 1819^13 mod 2537 , 1415^13 mod 2537 ) = ( 2081, 2182 ).
      To decrypt we use d = 937 and compute
      ( 2081^937 mod 2537 , 2182^937 mod 2537 ) = (1819,1415)

  We will show that f_enc and f_dec are inverses of one another shortly.
  But first, why is f_enc() easy and f_dec() hard to evaluate?
    f_enc() requires multiplying by M and taking the remainder mod n,
      both of which are easy, even if M and n are large.
    f_dec() equally easy if we know d, which only the Receiver knows.
      Why is it hard to figure out d? All you have to do is
       1) factor n=p*q
       2) use Euclidean algorithm to compute d so d*e ==1 mod (p-1)*(q-1)
```

```
  But 1) is very hard: Best algorithms would take billions of years
  if n has 400 digits. And any other known algorithm to compute d
  leads to computing p and q too. So quality of encryption depends on
  large integers being very hard to factor. If you figure out an
  algorithm to factor quickly, you can become rich or famous.

Proof that f_dec() is inverse of f_enc requires
Fermat's Little Theorem (proof is questions 15-17 in section 2.6):
    If p is prime and p /| a, then a^(p-1) == 1 mod p

Proof that f_dec(f_enc(M)) = M, where M < p,q
f_dec(f_enc(M)) = f_dec(M^e mod n) = (M^e)^d mod n = M^(e*d) mod n.
We need to show that M^(e*d) mod n = M mod n = M, since M < p*q = n.
Now e*d == 1 mod (p-1)*(q-1) so e*d = 1+m*(p-1)*(q-1) for some m. Then
M^(e*d) mod n = M^(1 + m*(p-1)*(q-1)) mod n
              = M * M^(m*(p-1)*(q-1)) mod n
Now since M < p and M < q, and p and q are prime, we must have
gcd(M,p) = gcd(M,q) = 1. Then Fermat's Little Theorem implies that
M^(p-1) == 1 mod p and M^(q-1) == 1 mod q.
Thus  M^(e*d) = M * (M^(p-1))^(m*(q-1))
             == M * (1)^(m*(q-1)) mod p
             == M mod p
and  M^(e*d) = M * (M^(q-1))^(m*(p-1))
            == M * (1)^(m*(p-1)) mod q
            == M mod q.
Finally, by the Chinese Remainder Theorem, M^(e*d) is the unique
solution mod p*q to
  x == M mod p
  x == M mod q
so M^(e*d) mod n = M as desired.

For RSA to be useful, we need to find a lot of large primes.
We will not discuss the algorithm for finding them, but just
discuss the theorem that says there are a lot to be found:

Def: pi(n) = the number of primes <= n
Ex:  pi(20) = |{2,3,5,7,11,13,17,19}| = 8
Theorem (Prime Number Theorem): The limit as n -> infinity of
   pi(n) / (n/ log_e n) = 1

EX:   n     pi(n)   n/log_e(n)    pi(n)/ (n/log_e n)
     10^1      4       4.3         .92
```

```
     10^2      25      21.7       1.15
     10^3     168     144.8       1.16
     10^4    1229    1085.7       1.13
     10^5    9592    8685.9       1.10
     10^6   78498   72382.4       1.08
     10^7  664579  620420.7       1.07
     10^8 5761455 5428681.0       1.06
```

The point is that the ratio in the last column is slowly approaching 1

So about what fraction of 200 decimal digit numbers are prime?

```
   # 200 digit primes / # 200 digit numbers
=  ( pi(10^200) - pi(10^199) ) / (10^200 - 10^199 )
~  ( 10^200/log_e(10^200) - 10^199/log_e(10^199) ) / (10^200 - 10^199)
~  .002 or about 1 out of 500
```

So if you pick 500 random 200 digit numbers,
there is a reasonable chance that one is prime