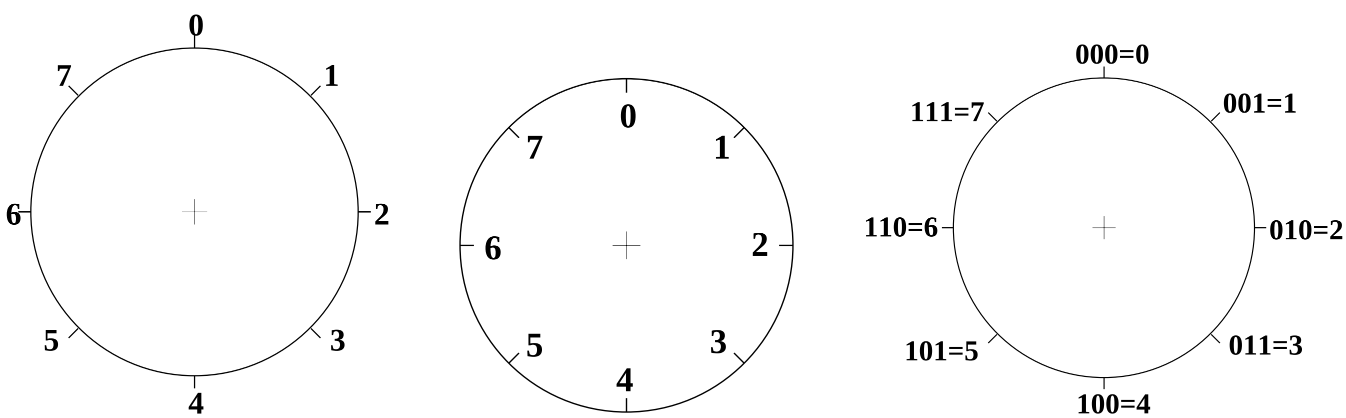
```
Math 55 - Spring 2004- Lecture notes # 8 - Feb 12 (Thursday)

   Recall that there will be midterm Tuesday, Feb 17, covering
      up to section 2.3, plus definition of a mod b

   Closed book, notes, computer, calculator, ...

   To discourage cheating, we will have many versions of midterm

   Goal for today: Recall definition of division algorithm, mod
                   Application: Modular arithmetic
                   Application: How computers do integer arithmetic

    Theorem (division algorithm) given integers a, d>0 (divisor), there is a
        unique q (quotient) and r (remainder) such that 0<=r<d, a=q*d+r
    DEF if a and d>0 are integers as above, then a mod d = r,
        remainder after dividing a by d

 Application of Division Algorithm: Modular Arithmetic

 DEF We say "a is congruent to b mod d" (or "a == b mod d") if d|(a-b)
     Otherwise we say "a !== b mod d"
 EX: 3 == 17 mod 7 because 7 | (17-3)
 Thm 1: a == b mod d if and only if there is an integer k such that
        a = b + k*d
        proof (=>) a == b mod d -> d|(a-b) -> exists k: a-b=k*d -> a=b+k*d
        proof (<=) a=b+k*d -> a-b = k*d -> d|(a-b) -> a == b mod d
 EX: 3 == 17 mod 7 <=> 7 | (17-3)  <=>  17=3+2*7
 Thm 2: a == b mod d if and only if a mod d = b mod d:
        proof (<=) a mod d = b mod d -> a=qa*d+r, b=qb*d+r, ->
                   (a-b)=(qa-qb)*d, -> d|(a-b) -> a == b mod d
        proof (=>) a == b mod d -> a = b + k*d by Thm 1, so when dividing
                   a = qa*d+ra, b=qb*d+rb, with 0 <= ra,rb < d, we get
                   ra-rb = (a-qa*d)-(b-qb*d) = (a-b)+qb*d-qa*d = (k+qb-qa)*d
                   now -d < ra-rb < d and ra-rb is also a multiple of d,
                     so ra-rb=0
 EX: 3 == 17 mod 7 <=> 3 mod 7 = 17 mod 7
ASK&WAIT: is 111 == 63 mod 3? is 123 == 6789 mod 2?
 Thm 3: a==b mod d and c==e mod d => a+c == b+e mod d
        proof: a = qa*d+r1 and b = qb*d+r1 and c = qc*d+r2 and e = qe*d+r2 ->
               (a+c)=(qa+qc)*d+r1+r2 and b+e=(qb+qe)*d+r1+r2 ->
               (a+c)-(b+e) = d*(qa+qc-qb-qe) -> d|(a+c-b-e) -> a+c == b+e mod d
```

```
 EX: 3==17 mod 7 and 11==4 mod 7 => 3+11 == 17+4 mod 7  (7|(21-14), i.e. 7|7)
ASK&WAIT: Is 112+227 == 31+65 mod 3?
 Thm 4: a==b mod d and c==e mod d => a*c == b*e mod d
        (try to prove this yourself)
 EX: 3==17 mod 7 and 11==4 mod 7 => 3*11 == 17*4 mod 7  (7|(68-33), i.e. 7|35)
ASK&WAIT: Is 112*227 == 31*65 mod 3?

 DEF: "arithmetic modulo d" or "modular arithmetic with modulus d" means
      doing integer arithmetic (+,-,*) where any two a,b satisfying
      a == b mod d are considered the same, because we only care what
      the answer equals mod d
 Thm 3 means that if we want to add and subtract numbers mod d, we can
      take any number or intermediate result and add a multiple of d to it
      (replace it by its value mod d) without changing the final answer.
 Thm 4 says the same thing about multiplication
 Ex: (13+15)*(2+8) mod 8 can be computed the following equivalent ways:
     (1) ((13+15)*(2+8)) mod 8 = 280 mod 8 = 0 mod 8
     (2) 13+15==28 mod 8 == 4 mod 8 and 2+8==10 mod 8 ==2 mod 8 so
         ((13+15)*(2+8)) mod 8 == 4*2 mod 8 == 8 mod 8 == 0 mod 8
ASK&WAIT: Any other ways?

  Here are several useful applications of modular arithmetic:
  Thm: Let x = d(n-1)d(n-2)...d(0) be an n-digit decimal integer.
       Then 3|x if and only if 3 | d(n-1)+d(n-2)+...+d(0), i.e. the
       sum of x's decimal digits.
   Proof. We want to show that x mod 3 = 0 if and only if
          d(n-1)+...+d(0) mod 3 = 0.
      We will show more, namely that x == d(n-1)+...+d(0) mod 3:
          x == sum_{i=0 to n-1} d(i)*10^i  mod 3 ... by def of decimal number
            == sum_{i=0 to n-1} d(i)*(10^i mod 3)  mod 3
            == sum_{i=0 to n-1} d(i)*(10 mod 3)^i  mod 3
            == sum_{i=0 to n-1} d(i)*(1)^i  mod 3
            == sum_{i=0 to n-1} d(i) mod 3
          as desired
   Thm: Let x = d(n-1)d(n-2)...d(0) be an n-digit decimal integer.
       Then 9|x if and only if 9| d(n-1)+d(n-2)+...+d(0), i.e. the
       sum of x's decimal digits.
    Proof: the same as above, substituting 9 for 3
ASK&WAIT: What about a rule for deciding if 11 | x?
ASK&WAIT: what about a rule for deciding if 7 | x?

 EX: Simplest way to implement "arithmetic modulo 8" means
```

only use numbers in set S={0,1,2,3,4,5,6,7};
after every operation (like 3*4) take result modulo 8 to get
number (4) in S

EX: Take a disk with d=8 equispaced points on circumference, labelled
0 (at top), 1 (to right) and around to d-1 = 7. Take another similar
disk with same center. to add a+b, align 0 of second disk
with a of first disk, and see where b of second disk hits
first disk, namely at a=b mod d ("circular slide rule").
Multiplication can be thought of as repeated addition.



EX: Computer Implementation of arithmetic mod 8:
"Unsigned Integers" in C, C++
represent numbers in base 2 (with 3 binary digits, or bits):

Addition is done as in elementary school: add bits from right
to left, where you get a "carry" from one column of bits to the
next if the sum in the column is at least 2 (10 in decimal), as
indicated below. If there is a carry out of the last column, we
ignore it, since there is no place to "carry it to".
This discarded carry represents an 8, so the final sum is 8 smaller
than the true value, eg 12-8=4 instead of 12, which is the
answer mod 8.

```
carries: 000              100               110               000
           001_2 = 1        010_2 = 2         101_2 = 5         100_2 = 4
         + 010_2 = 2      + 011_2 = 3       + 111_2 = 7       + 101_2 = 5
         -----------      -----------       -----------       -----------
           011_2 = 3        101_2 = 5         100_2 = 4         001_2 = 1
                                              = 12 mod 8        = 9 mod 8
```

EX: 2's complement arithmetic: how computers do integer arithmetic
with positive and negative integers.

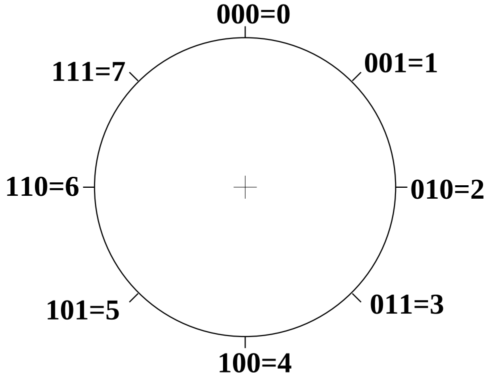
Suppose computer words had just 3 bits, representing 2^3=8 numbers. (A real computer would use 32 bits, representing 2^32 numbers, but it is the same idea.). Then instead of doing modular arithmetic on the set S={0,1,2,3,4,5,6,7}, we use the set S={-4,-3,-2,-1,0,1,2,3}. I.e. after each operation, we add a multiple of 8 (or 2^32) to the result to get an answer in S.

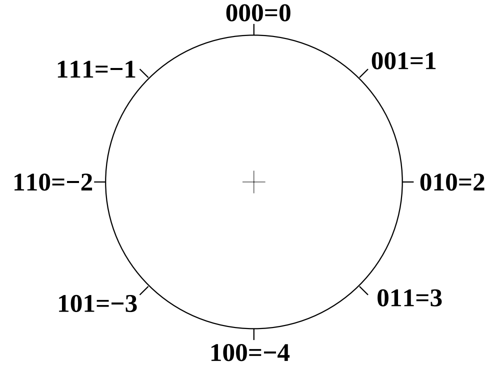
The way you tell positive from negative numbers among the 8 bit patterns on our 3-bit computer is to look at the leftmost bit, the "sign bit": it is 0 for nonnegative numbers, and 1 for negative.

bit pattern	unsigned integer	2's complement integer
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4 = 4-8
101	5	-3 = 5-8
110	6	-2 = 6-8
111	7	-1 = 7-8

Rule to interpret bit pattern as 2's complement number:
 if leading bit ("sign bit") is 0
     the number is positive, with the same value as the unsigned integer
 else if the sign bit is 1
     the number is negative, with the value of unsigned integer minus 8

Here are circular slide rules for unsigned (left) and
    2s complement (right) integer arithmetic:





With 3 bits, numbers range from -2^2   = 100_2   = -4

```
                                 to  2^2-1    = 011_2    =   3
      With 32 bits, numbers range from -2^31     = 10...0_2 = -2147483648
                                 to  2^31 - 1 = 011..1_2 =  2147483647
   Arithmetic is done the same way, whether unsigned or 2's complement:
   carries: 000              100               110               000
            001_2 = 1        010_2 = 2         101_2 = -3        100_2 = -4
          + 010_2 = 2      + 011_2 = 3       + 111_2 = -1      + 101_2 = -3
          -----------      -----------       -----------       -----------
            011_2 = 3        101_2 = -3        100_2 = -4        001_2 = 1
                             = 2+3 - 8                           = -7 + 8
ASK:  int a,b,c,d,e,f on 3 bit machine
      a = 3
      b = a+1                ... what is b?
      c = a+2                ... what is c?
      d = 2
      e = 2*d                ... what is e?
      f = 2*e                ... what is f?
ASK:  int a,b,c,d,e,f on 32 bit machine (like most)
      a = (2^30-1) + 2^30    ... what is a?
      b = a+1                ... what is b?
      c = a+2                ... what is c?
      d = 2^30               ... what is d?
      e = 2*d                ... what is e?
      f = 2*e                ... what is f?
Try running above program on your own machine
```