CS100J 09 March, 2006
More on Loops
Reading: Secs 7.1–7.4

A Billion. The next time you hear someone in government rather casually use a number that includes the word "billion", think about it.

- A billion seconds ago was 1959.
- A billion minutes ago Jesus was alive.
- A billion hours ago our ancestors were living in the Stone Age.
- A billion days ago no creature walked the earth on two feet.
- A billion dollars lasts 8 hours and 20 minutes at the rate our government spends it.

On "fixing the invariant"

```
// {s is the sum of 1..h}
s= s + (h+1);
h= h+1;
// {s is the sum of 1..h}
```


On "fixing the invariant"

```
// {s is the sum of h..n}     s = 5 + 6 + 7 + 8      h = 5, n = 8
s= s + (h-1);
h= h-1;
// {s is the sum of h..n} s = 4 + 5 + 6 + 7 + 8      h = 4, n = 8
```


Loop pattern to process a range m..n–1
(if m = n, the range is empty)

```
int h= m;
// invariant: m..h–1 has
//          been processed
while (h != n) {
    Process h;
    h= h+1;
}
// { m..n–1 has been processed
}
```

```
// invariant: m..h–1 has
//          been processed
for (int h= m; h != n; h != d+1) {
    Process h;
}
// { m..n–1 has been processed }
```

5..4

5..5

5..6

5..7

Loop pattern to process a range m..n
(if m = n+1, the range is empty)

```
int h= m;
// invariant:  m..h–1
//    has been processed
while (h != n+1) {
    Process h;
    h= h+1;
}
// { m..n has been processed }
```

```
// invariant:  m..h–1
//    has been processed
for (int h= m; h <= n; h= h+1) {
    Process h;
}
// { m..n has been processed }
```


Decimal	Octal	Binary
001	001	1 = 2**0
002	002	10 = 2**1
003	003	11
004	004	100 = 2**2
005	005	101
006	006	110
007	007	111
008	010	1000 = 2**3
009	011	1001
010	012	1010
011	013	1011
012	014	1100
013	015	1101
014	016	1110
015	017	1111
016	020	10000 = 2**4
032	026	100000 = 2**5
…		
256	400	100000000 = 2**8

2**n in binary is:

1 followed by n zeros.

2**15 is 32768 (in decimal).

n is called the (base 2) *logarithm* of 2**n.

The log of 32768 = 2**15 is 15.

To translate an octal number into a binary number, just translate each of digits:

octal 745

is

binary 111100101

Logarithmic algorithm to calculate b**c, for c >= 0

```
/** = b**c, given c ≥ 0 */
public static int exp(int b, int  c) {
   if (c ==  0) return 1;
   if (c%2 = 0) return exp(b*b, c/2);
   return b * exp(b, c–1);
}
```

Rest on identities:

b**0 = 1

b**c = b * b**(c-1)

for even c, b**c = (b*b)**(c/2)

3*3 * 3*3 * 3*3 * 3*3 = 3**8

(3*3)*(3*3)*(3*3)*(3*3) = 9**4

Algorithm processes binary representation of c

Suppose c is 14 (1110 in binary)

1. Test if c is even: test if last bit is 0

2. To compute c/2 in binary, just delete the last bit.

Algorithm processes each bit of c at most twice.

So if c is 2**15 = 32768, algorithm has at most 2*15 = 30 recursive calls!

Algorithm is *logarithmic in c*, since time is proportional to log c

7

Iterative version of logarithmic algorithm to calculate b**c,
for c >= 0 (i.e. b multiplied by itself c times)

```
/** set z to b**c, given c ≥ 0 */
int x= b; int y= c; int z= 1;
// invariant:  z * x**y = b**c  and 0 ≤ y ≤ c
while (y != 0) {
   if (y % 2 == 0)
        { x= x * x; y= y/2;  }
   else { z= z * x; y= y – 1; }
}
// { z = b**c }
```

Rest on identities:

b**0 = 1

b**c = b * b**(c-1)

for even c, b**c = (b*b)**(c/2)

3*3 * 3*3 * 3*3 * 3*3 = 3**8

(3*3)*(3*3)*(3*3)*(3*3) = 9**4

Algorithm is *logarithmic in c*, since time is proportional to log c

8

Iterative version of logarithmic algorithm to calculate b**c,
for c >= 0 (i.e. b multiplied by itself c times)

```
/** set z to b**c, given c ≥ 0 */
int x= b; int y= c; int z= 1;
// invariant:  z * x**y = b**c  and 0 ≤ y ≤ c
while (y != 0) {
   if (y % 2 == 0)
        { x= x * x; y= y/2;  }
   else { z= z * x; y= y – 1; }
}
// { z = b**c }
```

Rest on identities:

b**0 = 1

b**c = b * b**(c-1)

for even c, b**c = (b*b)**(c/2)

3*3 * 3*3 * 3*3 * 3*3 = 3**8

(3*3)*(3*3)*(3*3)*(3*3) = 9**4

Algorithm is *logarithmic in c*, since time is proportional to log c

9

Calculate quotient and remainder when dividing x by y

x/y = q + r/y **21/4= 4 + 3/4**

Property: x = q * y + r and 0 ≤ r < y

```
/** Set q to  and r to remainder.
    Note: x >= 0 and y > 0 */
int q= 0; int r= x;
// invariant:  x = q * y + r    and 0 ≤ r
while (r >= y) {
        r= r – y;
        q= q + 1;
}
// { x = q * y + r    and   0 ≤ r < y }
```

10