7/13 again. 14/13 for first digit.

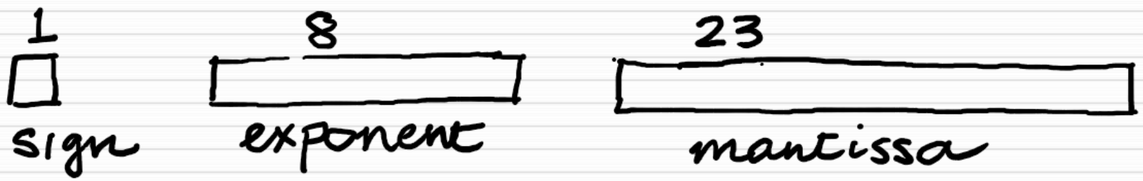
Repeat quotient remainder double

$\frac{14}{13}$	$\frac{2}{13}$	$\frac{4}{13}$	$\frac{8}{13}$	$\frac{16}{13}$	$\frac{6}{13}$	$\frac{12}{13}$	$\frac{24}{13}$	$\frac{22}{13}$	$\frac{18}{13}$	$\frac{10}{13}$	$\frac{20}{13}$	$14/13$ ...
1	0	0	0	1	0	0	1	1	1	0	1	
$1/13$	$\frac{2}{13}$	$\frac{4}{13}$	$\frac{8}{13}$	$\frac{3}{13}$	$\frac{6}{13}$	$\frac{12}{13}$	$\frac{11}{13}$	$\frac{9}{13}$	$\frac{5}{13}$	$\frac{10}{13}$	$\frac{7}{13}$	

.1000100111101 recurring

Yes, there was an error last Friday at $\frac{12}{13}$

Single precision "float"



Exponent BIASED so $b - 127 = e$

sign = 1 negative

The floating point rep. of a number $x \neq 0$
given its true value is $\pm 2^e 1.a_1 \cdots a_{24} \cdots$
is (a) sign (b) biased exponent (add 127)
(c) mantissa $a_1 .. a_{23}$ (rounded up if $a_{24} = 1$)
(d) 'little endian'

C programming, Monday 9/11/20

A floating-point number is an encoding of a
number of the following kind:
  (sign) M x 2^e, where 1 <= M < 2
This is related to `scientific notation' which you
can produce in C with %e format:

```
  printf("%e\n", 12345.678);
```

output

```
  1.234568e+04
```

meaning

```
 1.234568 x 10^4
```

single precision
 1 sign bit 8 exponent 23
mantissa

double
 1 sign bit 11 exponent 52
mantissa

Example Convert (-5.0)/104 to single precision
the sign is negative. Sign bit is 1 for negative.
The exact meaning of a 32-bit float,
partitioned as follows
(sign 1 bit) (`biased exponent' b) (`mantissa'
a_1 a_2 .... a_23)
is
(sign) (2^(b-127)) (1.a_1a_2 .... a_23)
except for zero where everything is zero.

Working with (-5.0)/104
We need first to multiply 5.0/104 by a power of 2 to bring it into the correct range for a mantissa.
That is, at least 1 and less than 2.
Well, if you multiply by 32 you get 20.0/13 which is in the correct range. That is 2^5.
Therefore
M x 2^(5) = 5.0/13. The correct exponent is -5.
Add it to 127 to get the biased exponent, 122.
Convert this to an 8-bit binary number (face value)

Or more quickly convert to hex. 122 = 16*7 + 10
7a in hex or 0111 1010 in binary. This is the biased exponent.

Example x : $\frac{-5}{104} = -\frac{5}{13\times 8}$ SIGN 1 (a)

Multiply $\frac{5}{104}$ by 2^5 : $\frac{20}{13}$ between 1 and 2.

So exponent e = -5 biased $127-5$

or in binary 01111111

-101

01111010 BIASED exp (b)

mantissa. $\frac{20}{13} = 1 + \frac{7}{13}$ DONE

$a_1 .. a_{24}$ = 1000 1001 1101 1000 1001 1101 round up

mantissa 10001001110110001001111

altogether

1 011 1101 0100 0100 1110 1100 0100 1111

b d 4 4 e c 4 f 4f ec 44 bd

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_10	a_11	a_12
1	0	0	0	1	0	0	1	1	1	0	1

So 20.0/13 =
1 . 1000 1001 1101
---------- recurring

Check:
1 + (2205/4096) (1/4096 + 1/4096^2 ...)
= 1 + 2205/4095 = 1 + 7/13 = 20/13.
Correct (sum geometric series)

24 bits of this infinite sequence
1000 1001 1101 1000 1001 110 1
The 24th bit is 1: ROUND UP.

Equivalently, add 1 to 23 bits
1000 1001 1101 1000 1001 110
+ 1
1000 1001 1101 1000 1001 111

sign biased exp mantissa
1 0111 1010 1000 1001 1101 1000 1001 111
1011 1101 0100 0100 1110 1100 0100 1111
b d 4 4 e c 4 f

One more complication
4f ec 44 bd little endian

Arrays and initialisation
int a[100]; double b[200];

a[0], a[1], ... , a[99] A PECULIARITY of C

Distinguish
int a[10]; --- 10 is the size of the array
printf("%d\n", a[5]) --- a[5] is the sixth
element of the array

Reading in an array from the keyboard
Look at section 8.1 of the web notes

```
#include <stdio.h>
main()
{ double a[1000]; // big enough?
   int count;  double x; count = 0;
   while ( scanf("%lf", &x ) == 1 )
   { if ( count < 1000 ) // Note!!!!
      { a[count = x;  count = count+1; }
}
printf("%d numers read\n",count);
int i;
for (i=0; i<count; i=i+1)
{  printf(" %f", a[i]); }
printf("\n");
printf("and  in reverse order\n");
for (i=count-1; i>=0; i=i-1)
{ printf(" %f", a[i]); }
} // end  of program
```

Initialisation

int x = 4; double y = 2·3;

(i) saves keystrokes

(ii) Risky because one might assume x==4 after it was changed

(iii) Very useful with arrays, especially for 'tables' which don't change.

(iv) a character string is an array of characters

char hello[6] = "hello";

char *monthname[12] = {"January", "February", "March", ...};

ONLY INITIALISATION

NO BULK ASSIGNMENTS

Initialising arrays, very convenient in C.
int month_length[12] =
{31,28,31,30,31,30,31,31,30,31,30,31};

Note (i) curly braces (ii) followed by a semicolon.
Another inconsistency in C.

Characters: `char' is used for declaring
character data, actually single bytes (groups of
8 bits)
char x; char y = 'z'; char newline = '\n'

Character strings: arrays of characters
char hello[6];

A character string is usually assumed to be formed of ASCII characters. Initialisation is possible:

```
char hello[6] = "hello";   NOTE 6 not 5
```

but assignment is NOT possible

```
hello = "hello";  is invalid
```

You cannot assign values directly to an array.
On the other hand

```
hello[1] = 'a'
```

is valid, and it changes the array to "hallo".

Why do you need 6 characters to hold a character string of length 5?

char hello[5] = {'h','e','l','l','o'};
is valid C code but is missing something vital: a marker for the end of the string. For that the null character '\0' is used (it does not represent any printable character).

char hello[6] = "hello";
and
char hello[6] = {'h','e','l','l','o','\0'}
are both correct and equivalent.

Useful example: char hex_digit[17]="0123456789abcdef";

Day in week calculation, for a day in this century.
dd mm yy
1<=mm<=12, 0<=yy<=99
add together
6
yy
yy/4
Total number of days in the months before the mm-th month
+ 6 if mm < 3 and yy % 4 = 0
And take the remainder modulo 7. This will give an integer from 0 to 6, with 0 for Sunday.