COMP20005 Workshop Week 9

Preparation: • open `grok`, `jEdit`, and `minGW` (or `Terminal` if yours is a `Mac`) • download this slide set (`ws9.pdf`) from github.com/anhvir/c205 if you like	
1 2 3	Discussion 1: Representation of integer Discussion 2: Representation of float Discussion 3: Numerical computation, root finding, bisection method, Ex 9.5
LAB	Exercises in: • number representation, • root-finding, • arrays . Note: most of today’s exercises are not in grok

Numeral Systems

214.39	2	1	1	.	3	9
Position	2	1	0	Dot	-1	-2
Value	2×10^2	1×10^1	4×10^0		3×10^{-1}	9×10^{-2}

→ *base* = 10 (decimal)

Other bases: binary (base= 2), octal (base= 8), hexadecimal (16)

$21.3_{(10)} = 2 \times 10^1 + 1 \times 10^0 + 3 \times 10^{-1}$

$1001_{(2)} = 1x2^3 + 0x2^2 + 0x2^1 + 1x2^0 = 9_{(10)}$

$5B_{(16)} = 5x16^1 + 11x16^0 = 91_{(10)}$

Note: hexadecimal uses 6 additional digits: A, B, C, D, E, F with the values 10-15

Converting between bases 2 and 16 is easy!

- Because 1 hexadecimal digit is equivalent to 4 binary digits.

```
110101101     →
     AFB5     →

111110011011  →
        3CD   →
```

Converting Binary → Decimal

Just expand using the base=2:

$$\ldots\ b_3\, b_2\, b_1\, \mathbf{b_0}\, .\, b_{-1}\, b_{-2}\ \ldots\ _{(2)}$$

is $\ldots\ b_3x2^3 + b_2x2^2 + b_1x\,2^1 + \mathbf{b_0} + b_{-1}x2^{-1} + b_{-2}x2^{-2} \ldots$

Examples: `1101` →

`1.011` →

Practical advise: remember

128	64	32	16	8	4	2	1	0.5	0.25	0.125
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}

Decimal→Binary: Integer Part

Changing integer x to binary: Just divide x and the subsequent quotients by 2 until getting zero. The sequence of remainders, in reverse order of appearance, is the binary form of x.

Example: 23

operation	quotoion	remainder
23 :2	11	1
11:2	5	1
5:2	2	1
2:2	1	0
1:2	0	1

So: $23 = 10111_{(2)}$ $11 =_{(2)}$ $46 =_{(2)}$

Decimal→Binary: Fraction Part

Fraction: Multiply it, and subsequent fractions, by 2 until getting zero. Result= sequence of integer parts of results, in appearance order. Examples:

0.375				0.1		
operation	int	fraction		operation	int	fraction
.375 x 2	0	.75		.1 x 2	0	.2
.75 x 2	1	.5		.2 x 2	0	.4
.5 x 2	1	.0		.4 x 2	0	.8
				.8 x 2	1	.6
				.6 x 2	1	.2

So: $0.375 = 0.011_{(2)}$ $0.1 = 0.00011(0011)_{(2)}$

Now try convert: 6.875 to binary

Exercise: Converting Decimal->Binary

- $7_{(10)} = ?_{(2)} = ?_{(16)}$
- $130_{(10)} =$
- $6.375_{(10)} =$
- $9.2_{(10)} =$
-

Representation of integers (in computers) using w bits

- Note that we use a fixed amount of bits `w`
- Make difference between `unsigned` and `signed` integers (`unsigned int` and `int` in C)

`unsigned` integers :

- Range: 0.. 2^w-1
- Representation: Just convert to binary, then add `0` to the front to have enough `w` bits.

Representation of integers (in computers) using w bits

- `signed` integer range: -2^w to 2^w-1

- To represent `signed` integers `x`:
  - Positive numbers: a `0-bit`, followed by the binary representation of `x` in `w-1` bits.
  - Negative numbers: using twos-complement of `x` in `w` bit. The first bit will always be `1`.

Finding twos-complement representation in w bits for negative numbers in 3 step

Suppose that we need to find the twos-complement representation of $-x$, *where* x *is positive, in* $w=16$ *bits. Do it in 3 steps:*

1) *Write binary representation of* $|x|$ *in* w *bits*
2) *Find the rightmost* `one-bit`
3) *Inverse (ie. flip 1 to 0, 0 to 1) all bits on the left of that rightmost* `one-bit`

find the 2-comp repr of **`-40`**	**Bit sequence**			
1) bin repr of 40 in 16 bits	0000	0000	0010	1000
2) find the `rightmost 1`	0000	0000	0010	1000
3) inverse its left	1111	1111	1101	1000

Exercise: 2-complement representation in w=16 bits

Q: What are `17`, `-17`, `34`, and `-34` as 16-bit twos-complement binary numbers, when written as (a) binary digits, and (b) hexadecimal digits?

Decimal	2-complement in binary digita	2-cmplement in hexadecimal digits
17		
-17		
34		
-34		

Representation of floats

- We learnt 2 different formats:
- one as described in `numericA.pdf` and in the text book
- another is an `IEEE standard`, which is:
  - employed in most of modern computers,
  - demonstrated in the lecture, and
  - you can find/experiment with using the program `floatbits.c  (numericA.pdf p.27)`.

Representation of floats (as described in numericA.pdf)

- `sign` `e` `m`

1 bit sign | w_e bits two-compl of e | w_m bits representation of first n bits of mantissa m

- Convert $/x/$ to binary form, and transform so that:

  $|x| = 0.b_0b_1b_2\ldots \times 2^e$ *where* $b_0 = 1$

- `e` is called exponent, $m = b_0b_1b_2\ldots$ is called mantissa
- x is represented as the triple (`sign, e, m)` as shown in the diagram.

Representation of 32-bit float: (IEEE 754, as in floatbits.c)

- w_s=1, w_e=8, w_m=23 $|x|=$ $1.b_1b_2\ldots$ x 2^e [Note the *different*
- *transformation* of $|x|$]
- ----------------------------------Bit Position --
- 1   2   9   10   32

- Sign ----------------------------------- --
- e+127 b_1 b_2 . . . $b_{22}b_{23}$

- That is:
- The sign bit is 0 or 1 as in the previous case
- e is represented in excess-127 format, which means e is represented as the unsigned value e+127 in w_e bits
- The first bit of the mantissa is omitted from the representation, and the mantissa is just $b_1b_2\ldots b_{23}$

- **Note:** Valid e is -126 → +127, corresponding to values 1 →254 . Value 0 used for representing 0.0, value 255 used to represent infinity. And, zero is all 32 zero-bit, and infinity is all 32 one-bit.

Representation of 32-bit float: (IEEE 754, as in floatbits.c)

- w_s=1, w_e=8, w_m=23 $|x| = 1.b_1 b_2 \ldots \times 2^e$
- Example: x= 3.5
- In binary: x= 11.1= 1.11 x 2^1

➔ sign bit: 0

➔ e=1 is represented as e+127= 128 in 8 bits

➔ e is represented as 1000 0000

➔ mantissa: 110 0000 0000 0000 0000 0000

➔ Final representation:

- 0100 0000 0110 0000 0000 0000 0000 0000
- or 4 0 6 0 0 0 0 0 (16)

Numerical Computations

`int`, `float`, `double` all have some range:

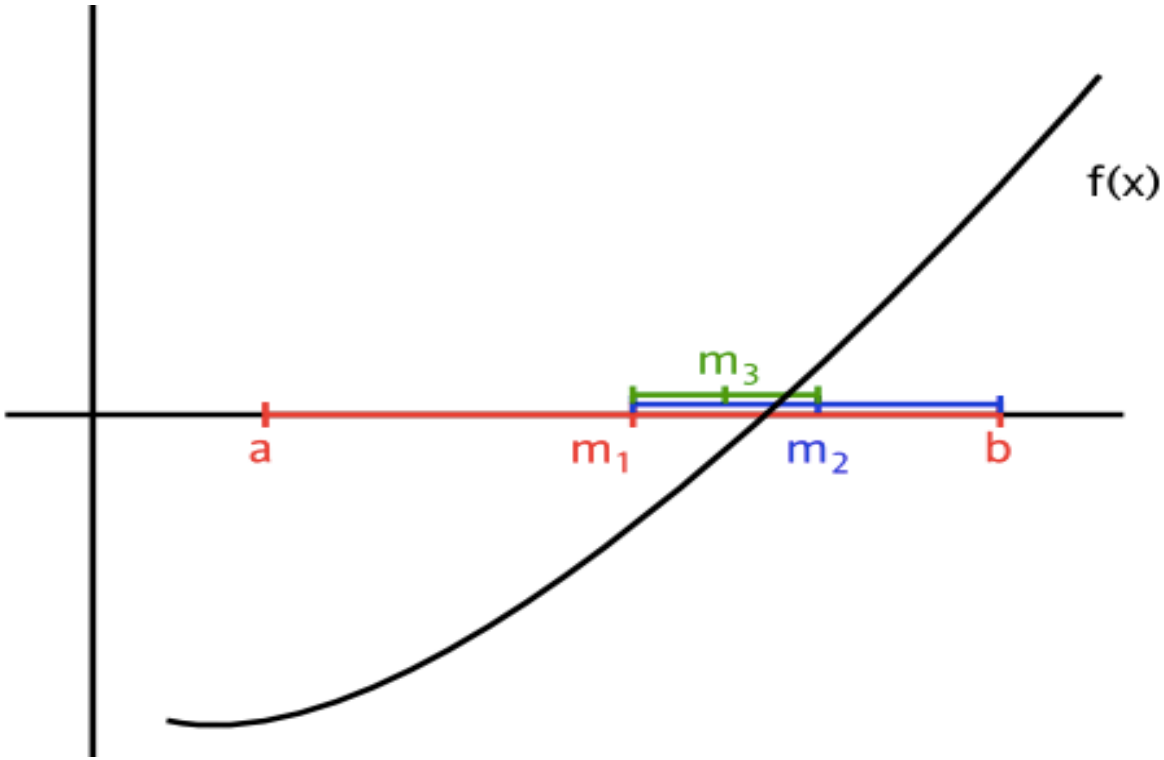
- `int` 32 bits: about -2×10^9 .. 2×10^9
- `int` 64 bits:
- `float` (w_s=1, w_e=8, w_m=23)
- `double` (w_s=1, w_e=11, w_m=52)

computation on `float`/`double` is imprecise, so

- use `if (fabs(x) < EPS)` instead of `if(x==0)`
- `if (fabs(a-b) < EPS)` instead of `if(a==b)`
- avoid adding a very large value to a very small value

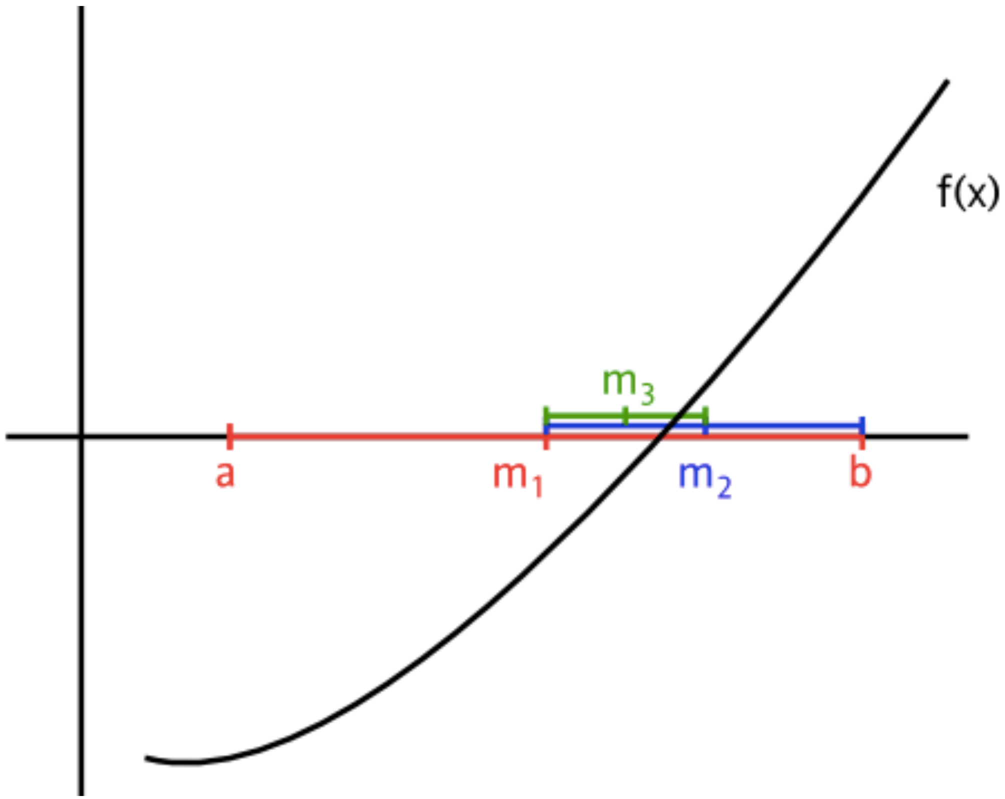
Root finding: see from p.27 of `numericA.pdf` for methods such as: *bisection*, *false position*, *fix-point iteration*, *Newton-Raphson*, *secant*

Roof Finding for f(x)=0 using the bisection method



Roof Finding for f(x)=0: bisection method

Ex. 9.5: The square root of z is the of equation $f(x) = x^2 - z$.
Evaluate the bisection method *by hand* (well, you can use calculators), start with a= 1 and b= 3. Stop when the length of the interval is 0.1 or less.



Lab: representation, root finding, arrays

1. Number representation:
   - Exercises **13.1**, **13.2** (provided in next page)
   - Understanding IEEE float representation (**pp. 14-15** this document), playing with `floatbits.c` (link provided in **p. 27** ← `numericA.pdf` ← **LMS.Module.Week9**)
2. Bisection method: exercise 9.5 (see previous page)
3. Re-examine the `cube_root()` function on page 77 of the textbook, `croot.c` (you can copy it from a link provided in **LMS.Module.Week9**). What method does it use? Explore what happens if: (a) very large numbers are provided as input; (b) very small (close to zero) numbers are provided; and (c) `CUBE_ITERATIONS` is made larger or smaller.
4. *Implement not-yet-done exercises from* `grok W7X, W8X`
5. **Ex. 9.8:** Suppose you have to write a function to return the k-th largest value in an array of n integers. What problem solving techniques might be used? Sketch, for each of the possible techniques, an algorithm for determining an answer to the problem.

Exercises 13.1, 13.2

13.1: *Suppose that a computer uses w= 6 bits to represent integers. Calculate the two-complement representations for 0, 4, 19, -1, -8, and -31; Verify that 19-8 = 11;*

13.2: *Suppose* w_s=1, w_e=3, w_m=12, *what's the representation of* 2.0, -2.5, 7.875 ?

Function `cube_root` in `croot.c`

```
#define CUBE_LOWER 1e-6
#define CUBE_UPPER 1e+6
#define CUBE_ITERATIONS 25

double cube_root(double v) {
   double next=1.0;
   int i;
   if (fabs(v)<CUBE_LOWER || fabs(v)>CUBE_UPPER) {
      printf("Warning: cube root may be inaccurate\n");
   }
   for (i=0; i<CUBE_ITERATIONS; i++) {
      next = (2*next + v/(next*next))/3;
   }
   return next;
}
```

What method does it use? Explore what happens if: (a) very large numbers are provided as input; (b) very small (close to zero) numbers are provided; and (c) CUBE_ITERATIONS is made larger or smaller.