Week

2

Numbers:

Look up Freeman Dyson

Natural = {1, 2, 3 ...} $\mathbb{N}$

Int = {... -3 _ 0 ... 3} $\mathbb{Z}$ +/-

Q ≈ # / by other #s

- 0 → neither positive nor negative
- computers do arithmetic in <u>finite fields</u>

unsigned #s range = 2x as big as signed (ex: 5 → -4 5 ↔ 4 5, us → 0 ↔ 9 0)

Half bits for ⊕ / half for ⊖

Every # of $\mathbb{N}$: THINK ABOUT THIS

$$a_k b^k + a_{k-1} b^{k-1} + \ldots + a_1 b^1 + a_0 b^0$$

b = base, $a_i \in \{0, \ldots, b-1\}$

Ints in C:

unsigned } short / long / long long } int / unsigned { 8-bit int → char

char → usually 8 bits
short → usually 16
int → at least 16
long → at least 32
long long → prob 64

<include stdint.h>

u int # _ t

u ↑ signed or us

↑ of bits

Binary Arithmetic:

LOOK BACK @ 12 NOTES

Add
0+0=0
1+0=1
1+1=10 (0, carry the 1)
} exclusive or

Mult
0×0=0
1×0=0
1×1=1
} AND

Do 101+11
101×101
ON QUIZ

Additive inverse = $i + \bar{i} = 0$
Mult. = $i(1/i) = 1$

LOOK BACK @ 12 NOTES

2's Complement Arithmetic:

TRY THIS

To get additive inverse:
Flip bits & add 1

Real #s: infinite
- Continuous / includes $\mathbb{Z}$ / $\mathbb{Q}$ / $\mathbb{R} - \mathbb{Q}$
(ints / rational / irrational)

Floating Point #s:
- Approximation (subset of $\mathbb{Z}$ / $\mathbb{R}$ / $\mathbb{Q}$)

float → 32 (single precision), double → 64, long double → 128

Decimal / Binary Fractions:
Program Mantissa @ 30 min
- calculates fractional part of binary num

UNDERSTAND

MESS AROUND

JOIN ACM IEEE

Single Precision: 32:20 formula for calculate value of floating point

sign = 1 bit

exp = 8 bits, precision = 23 bits

Double: sign = 1, exp = 11, prec. = 52 bits

long double = 15 bit exp, 112 bit mantissa

33:50 → compute π UNDERSTAND THIS

Big endian → big → high address → right

Little endian → # → low address → left

- use code on 37:18 for asgn 5

% (ints only) → remainder

Type promotion → lower types can be → no precision loss to higher types

Computers can only respresent a small subset of the integers

Check man pages

Numerical Computation: + = exclusive or

* = shift + addition / ÷ = shift + subtraction

Taylor Series

8 min → computing e^x (cannot use this)

↑ go back and watch

$\sqrt{x}$ @ 9:48

?

Look up ternary operator

Mathematica → terminal prof uses

Don't use equality testing with floating point #s

Floating point arithmetic:

- errors with rounding

depend which you use, diff. size ↓

To compare FP #s: check absolute / relative error

Arrays:

- no protection if you go off end of array (ex. @ index n+1)

Matrices

- m[3][3]; → m[0][0] to m[2][2];
- stored in row major order

Ex: int m[2][2] = {{1,2}, {3,4}};

- in memory (1D), rows are put sequentially

Matrix Multiplication (wow!)

name of the array is a pointer to 1st element

& (address of) operator → memory location of var

sizeof - # of bits of a type of variable
may not work for arrays / strings
add. of array → 8 bits

alias → two variable refrence the same memory

- arrays + pointers are equivalent in C

Dynamically Allocated Array!

free(a) ⟶ do after dynamic allocation

- matrix is an array of pointers

Go through it!!!

READ C BOOK!

Searching:

- Best case is finding element on first try

Bubble Sort!!!

If array is ordered, you find extrema in constant time

Binary Search

ordered array: you can find element in array $\log_2$(array size) time

Strings:

null character ↓ '\0'

- array of characters that ends with '0'

comparing: s-t

<0 - s<t

>0 - s>t

$=0$ - s==t

- must check bounds (not allocate too much/little memory)