ssh - secure shell

Git → track files
- can go back to earlier versions

Type vimtutor
↑
into terminal

Readme
- what it does
- how to build
- how to run
- errors

Binary arithmetic
Bits (manip.)
Addresses
Pointers

Kollected Code

Vicious ← BOOL

Latech
↑
for drawing pics
on
overleaf

git add design.pdf to computer
git pull to VM

eMacs
vi
VScode / Sublime Text
↑
text editor

GOOGLE
Markdown
syntax

<stdio.h> → standard I/O package

printf → print, 0 → success (return)

cc -o hello hello.c ← compile (not clang)

./hello ← run, // → comments

look in MAN printf ← %3.0%f 6.1f\n

EOF end of file

Use for loop when you can

Need to declare type of var and initiliaze (char → small ints) ' ' ← char

Variables of same name cause problems

can read from one file to another

International Obfuscated C Competition

define CONSTANT number

↑ processor when you put CONSTANT, number will be in that place

Python → interpreter (slower)

C → compiles → asm → machine code

Use clang to compile

Makefile → has rules on how to compile

Make sure code works on Linux SOMEHOW

XKCD ↑ comics

SIGN UP FOR TUTORING ON TUTORTRAC !!!

Booleans:

AND	$\wedge$	conjunction	&&
OR	$\vee$	disjunction	||
NOT	$\neg$	negation	!

inclusive

} in C

XOR $\oplus$ $A \oplus B = (A \vee B) \wedge !(A \wedge B)$

$A \oplus B = (A \wedge !B) \vee (!A \wedge B)$

- one or the other, not both

$A \oplus A = 0$, $A \oplus 0 = A$, $A \oplus 1 = !A$

XOR	T	F
T	F	T
F	T	F

CHECK

- nonzero ⇒ TRUE

0 = FALSE (only thing that is)

- logical expressions have type int

```
#include <stdbool.h>
scanf("%d", &n)
```

Short circuit eval. → when result of bool expr. is known, evaluation will stop

Switch():

```
switch(___){
    case 0:
    case 1:
    default:
}
```

use break after statement

- goto could also be used

DON'T

Loops:

@ 9:55 in lecture — tip for asgn2

while()

- top test loops (test evaluated before body)
- evaluated 0 or more times

for()

- top test loop
- parts → init., test, increment
- i = control variable (disappears after execution)
- init. and test don't need same variable

do {
} while ()

- execute at least once
- bottom-test loop

use for pig if u can switch statements are good for rolling dice

infinite loops

- to end, use break

  Ex: while(1), for(;;)

- could use goto for major errors

= assignment, == equality

Continue: like break, but it goes back to the beginning of the loop

LOOK MORE INTO SWITCH

Functions:

LOOK UP CALL BY NAME

- may or may not (void) return a value
- defined once / declared & call >1 times
- support functions before main
- main() run when program starts

```
return type func_name (parameters) {
    // declarations / assignment
}
```

- return type can be void / anything except array
- no parameters = VOID
- declared variables inside func. = local
- use snake case → (_ _ _) lowercase

Parameters:

- call-by-value → take value of parameter, make a copy of x, and give it to the function (x is not modified @ end)

formal: name of par. as used in func. body

actual: value supplied/passed copied to formal parameters

LOOK BACK @ CALL BY REFERENCE SLIDE

GO BACK TO ∫ PART

call by reference → can be emulated using pointers (addresses are passed as arguments)

function prototype:

return type func_name (parameters); ← or just types

- declaration of function needed if functions not defined before main

Macros in C use text replacement

#include → pastes code of given file into current file

#define → defines macro for program replace

#ifndefn / #out → LOOK

Headers:

- contain func. declarations / macros / etc.
- ends in .h

extern: write def later

static: variable persists across function calls
only visible inside that file

utility functions: Func. that do small things

Recursion: take factoring code & run it

HEADERS 10:08

MAKE COPY OF PUBLIC SSH KEY and PUT IN FILE

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)

Computational Complexity:

- Run times of algorithms matter

For loop going through all terms = $O(n)$

Look for loops for order

insertionSort → $O(n^2)$

only const. changed
$O(n) + O(n) = O(n)$

Loops next to each other → add complexity

Nested loops → multiply ex: $O(n2) \rightarrow O(n3)$

Depending on overhead (c), some algorithms are more effecient than others

Factorial → $O(n)$, Fibonacci → $O(2^n)$

Look over photo of algorithm

Recursion:

tail recursion?

- defined in terms of itself

Function call creates stack frame

- mem. access is slow

Cost → storage added to stack / copying memory

- Use recursion when it is natural

Why?

Binary Search → O(log(n)) with sorted array

- Loop inside loop → n^2 (loop inside divides by 2 → logn)

Better to pay cost once than many times to solve problem

Strings end with a null '\0'

Knight's Tour: ← ← RUN THIS CODE

RUN QUEENS CODE

10/27:

Bit Vectors & Sets:

Bit vectors:

Arithmetic left shift: 0's are shifted on the right

Bitwise operators:

& = AND, | = OR, <</>> = left/right

- Want to use shift with unsigned values

AND - to find value of a bit (preserve)

OR - to set value of a bit

XOR - to flip values? / NOT - opposite

Sets:

- well-defined unordered collections
- membership, delete, union, etc.

0 = not member, 1 = member

Comp.: NOT

Intersection = AND, U = OR, Diff = Complement AND

Set.c:

mask = 63 (in hex)

10/29:

GIT PULL RR

Entropy calculator in Comments REPO

Data Compression:

Channel - medium through which signal is transmitted

Transmitter -

Reciever - Decompress/decrypt/error correction

PROBABILITY:

Entropy → Uncertainty

$\log(1) = 0$

- Sum of probabilities = 1

$H(p_1, p_2, \ldots n)$ — probability of event p_i happening

- H is continuous in p

prob. of event occuring

$1 \leq p_i \leq n$

Formula: $H = -\sum_{i=1}^{n} p_i \log_2(p_i)$

Message 3:

ABCD $p_i(A) = \frac{1}{4}$, $p_i(B) = \frac{1}{4}$, $p_i(C) = \frac{1}{4}$, $p_i(D) = \frac{1}{4}$

$H = -\sum_{i=1}^{n} p_i (\log)_2 p_i$

$p_i = \frac{1}{4}$

$= 2$ (need 2 bits to show this

Run-length coding: (repeated data)

- Represent message with sequence of identical symbols

Huffman Coding:

- Assign each symbol a unique bit-string code
- Histogram
- Priority Queue (root → higher probrity / lower frequency)

@10 discussion of Heap

push → take parent and do thing

pop → to take off/put at top (Ax)

Build:
while q is not empty:
  pop
  pop
  join
  push

log(n) time b/c it's how many times you can half the heap b/c it's a binary tree

Code can be at MAX 256 bits long

Code table contains bit vectors

Rebuild tree by L = push, I = pop (with stack)

11/1:

Make: → build executables from source code

- Variety of command line flags/options
- Requires a Makefile

Makefile:
- Plaintext file (DAG)
- Composed of rules

Rule → target / dependencies / commands
nodes edges

Phony target:
- doesn't build exec. of file name (all / clean)

Make flags/options:
- -d: print debug info
- -f: specifies file to be read

Variables:
- Assignment (=/:=/?=/+=)

$(var_name) = value of variable

call by name

Lazy: = (wait to expand references until use)

Immediate: := (assignment in C)

call by value

Conditional: ?= (= assignment if var DOES NOT have value)

Concat: += (add extra text to defined var)

Dependencies:
- what is needed to build exec. (source code)

Command:
- Action to be executed (ex: clean/format)

Commands/Compilers/Compiler Flags:
CC: the C compiler to use
CFLAGS: list of compiler flags
OBJ: list of object files (.o)
Within rules:
$@: the name of target
$^: list of all dependencies for target
$<: name of first dependency

Shell: ??
- Command expansion

Wildcard ??

Patsubt (pattern substitution)
- Find words in file that matches pattern & replaces

* vs %:
- performs expansion
- pattern match placeholder

Recursive use of make:
look at slide
- can include within makefiles

11/10: Software Tools Philosophy

Scripting:

- Operate on higher level objects (files/processes)
- Reuse code/programs

Ex: hexdump / diff / awk to parse file

sh → first UNIX shell (wraps around UNIX, interface)
- Many shells exist (zsh, bash, csh)

BASH:

- command line interpreter
- Users can give input:
  - Interactively: as prompt/command line ($)
  - Batched: script ?

Terminal is a dumb thing

Commands: shell → interpreter

- Each line is a command, words split by whitespace

Ex: ls -a

xxd - binary files?

cat - prints contents of file

Types:
- alias (rename commands)
- builtins (bash-provided → printf, echo, cd)
- functions (sequence of commands to do task, takes arguments)
- executables (specified/executed by file path)

$PATH:
- look through directories to find file of name
- includes /bin + /usr/bin

Writing a script: #!/bin/bash (use bash shell)

- #! → magic number

Dotfiles: (hidden files)

.bashrc/.bashprofile

LOOK @ BASH BOOK IN RESOURCES

Pipes: "|"

- Connect stdout of one process to stdin of another
- chained together = pipeline

awk - program for parsing text

Variables/parameters:

- to store data (assign using =)
- Native type → strings
- Arrays doesn't need to be homogeneous
- expand parameters with $ (like * in C → dereference and get content
- can do arithmetic

Special parameters - $#/$?/$$/$_ (exit status; process ID; argument of last command)

Functions:

```
function name() {

}
```

For loop:

```
for f in *
```

If statement:

```
If [[test]]
```

- array → a list of strings (sparse array - gaps between indices)

Array access/usage on next slide

Globs/Patterns:

- glob: patterns used to match strings

  * - matches any string / ? → matches single character

- can test strings/integers/files (3 slides)
- LOOPS

File redirection - < > (direct to file/stdin/stdout)

11/12:

BIT PLAYER → AMAZON MOVIE claude shannon

Cryptography:

- take info and transform so only recipient can decrypt it
- Algorithm known, key is secret
- Data + key: key only known to authorized users

Plaintext → ciphertext → plaintext

$$P \rightarrow \underset{K_E}{E} \xrightarrow{C} \underset{K_D}{D} \rightarrow P$$

k = key E/D = encrypt/decrypt
p = plain c = cipher

Caesar cipher → uses modular arithmetic
- replaces letter w/ 3 letters down (modular shift)

Random + XOR = random + XOR = message?
- Unbreakable → use random key = same length as message

Simon cipher → IN COMMENTS REPO

Public-key cryptography:

Factoring large composites = hard

- Public = K_U, Private = K_R

$\Rightarrow E(K_U, E(K_R, M)) = E(K_R, E(K_U, M)) = M$

RSA → uses prime rings and exponentiation

Diffie-Hellman Key-Exchange = shared key (prime + mod)

11/19

Hashing / Bloom Filters:

- Take a key (string) and make a #
- # is uniformly distributed

Want to look up, insert, delete keys (find records of keys)

- You know exactly where it is → constant O(1) time

Limits → need to know size / waste memory

Hash tables:

- unordered collection of key-value pairs

Hash functions → takes key to generate indices (using hashing)

HT w/ m bits = h(key) → index $\in [0, m-1]$

÷ - based = $h(k) = k \bmod n$ (no * or +)

string = take chars, compute int, int % table size

Ex: take keys (mod 11) → store in HT indexed 0-10

Hash collision → 2 pieces of data have same hash value

Linear Probing: (Insertion)

- Move sequentially through HT until empty slot

If HT almost full, get close to O(n) time

Quadratic Probing:

- Square the sequence of hash values when deciding how far to look for next slot

Load factor = how many items compared with how many slots HT has

Double Hashing: LOOK BACK

Pure Storage

Chaining w/ Linked Lists:

Slot contains link (*) to linked list

If empty, new LL / If LL, append to it

Using tree instead: logn deep & balanced

Open addressing vs. Chaining

L/Q/D ProbHashing

LL/Trees
- bad cache performance
- extra space used

If deletions are allowed, don't use OA

Find/insert/delete

Bloom Filters: → internally like bit array

- data structure that tells you whether an element is present in set
- Operations: Add / Search

1 → element might be in set / 0 → element not in set

- Use multiple hash functions ↓ w/ more hash functions

Probability of false positives: ↑ w/ more items

Application: Cryptographic Hash

Ex Hash Function: (use speck)

Hash Function in Resources Repo

↑ ↑ ↑ ↑ ↑ ↑

11/22:

Regular Expressions: (Regex)

- Series of characters that define search pattern

Basic Syntax:

letter (matches character itself) / . (matches any character)

+ (match 1+ occurences) / * (match 0 occurences)

- Operations to search for characters

Theory of Finite Automata:

Regex → expressions that let you describe regular sets

DFA → single transition per letter

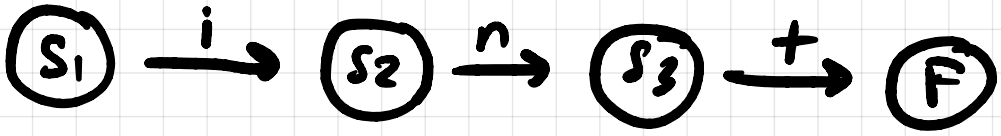
NFA → have many transitions per letter

Finite automata - abstract machine that recognizes patterns like strings

DFA example:

Regex: int input: "int"

He wrote a book on NFA???



NFA: simulating multiple states

Regex Package:

include <regex.h>

regex_t = NFA

Example of Regex in C