Introductory survey course
  no prerequisites
  basic principles of computer science
  learn to use computers effectively
  check FAQs on web

Topics introduced:
  hardware and software systems
  programming in C and other languages
  algorithms and data structures
  theory of computation
  applications to solving scientific problems

#include <stdio.h>
main()
{
    printf("This is a C program\n");
}

Q. How did the computer scientist die in the shower?
A. The instructions on the shampoo said "Lather, Rinse, Repeat"
Lecture Outline

C
  Basics
  Functions and Modules

MACHINE ORGANIZATION
  Toy Machine
  Machine Language
  Simulator

DATA STRUCTURES
  Data Types
  Data Structures
  Recursion
  Trees

ARCHITECTURE
  Boolean Logic
  Sequential Circuits
  Machine Architecture
Lecture Outline (continued)

AUTOMATA and LANGUAGES
  REs and FSAs
  Grammars
  Chomsky Hierarchy
  Turing Machines

PROGRAMMING SYSTEMS
  Memory Organization
  Modules and Libraries
  Objects

ANALYSIS OF ALGORITHMS
  Algorithms and Complexity
  Improving Performance
  NP-completeness

APPLICATIONS
  Graphics
  Mathematical Applications
  Perspective
Precepts

Fridays: prepared topics
  getting online
  C basics
  TOY programs
  C structures
  C pointers
  Postscript
  grep/awk
  C libraries
  Java
  Typesetting

Mondays: discussion and questions
  reading
  exercises
  programming assignments
COS 126 Survival Guide

Keep up with the course materials
read over handouts when you get them
www.CS.Princeton.EDU/courses/archive/fall97/c
prepare for precepts

Keep in touch
finger, mail
office hours
after class

Use the simplest tool that gets the job done

Understand your program
what would the machine do?
find the first bug
develop programs incrementally
plan multiple lab sessions

Ask for help when you need it
Linear feedback shift register

Machine consists of 11 BITS, or 0-1 values
Bit values change at discrete time points
Bit values at time T+1 completely determined by values at time T

<table>
<thead>
<tr>
<th>T</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>10^3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T+1</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

"XOR" of two bits is
  1 if they are different
  0 if they are the same

Magic properties:
  b^b = 0
  a^0 = a
  (a^b)^b = a^(b^b) = a^0 = a
LFBSR example

0 1 1 0 1 0 0 0 0 1 0
1 1 0 1 0 0 0 0 1 0 0
1 0 1 0 0 0 0 1 0 0 1
0 1 0 0 0 0 1 0 0 1 0
1 0 0 0 0 1 0 0 1 0 0
0 0 0 0 1 0 0 1 0 0 1
0 0 0 1 0 0 1 0 0 1 1
0 0 1 0 0 1 0 0 1 1 0
0 0 1 0 0 1 0 0 1 1 0
0 1 0 0 1 0 0 1 1 0 0
1 0 0 1 0 0 1 1 0 0 1
0 0 1 0 0 1 1 0 0 1 0
0 1 1 0 1 0 0 0 0 1 0

Bits "look" random (but aren't!)
Using “Random” Bits for Encryption

Convert message to bitstream

S E N D M O N E Y
100100010101100001000110101110011000010111001

Send bit-by-bit XOR with “random” bitstream

S E N D M O N E Y
100100010101100001000110101110011000010111001
00100110010001101010100001111010100011100101
10110111000110110001001010000100110001011100

Message looks random to anyone reading it

W ? M R E A F B Z

Receiver has identical machine
(Secretly) provide receiver with initial fill
Receiver computes XOR with SAME “random” bitstream

10110111000110110001001010000100110001011100

Works because \((a \oplus b) \oplus b = a \oplus (b \oplus b) = a\)
Properties of shift register “machine”

Clocked

Control: start, stop, or “load”
Data: initial values of bits

Built from very simple components
  “clock” (regular electrical pulse)
  electrically controlled shift register cell
    remembers value until clock “ticks”
  some wires “input”, some “output”

Scales to handle huge problems
  10 cells yields 1 thousand random bits
  20 cells yields 1 million random bits
  30 cells yields 1 billion random bits

BUT, need to understand abstract machine!
(higher math needed to know XOR taps)

Same basic principles used for computer
  clocked
  all built from switches with feedback
  control, data
  abstraction aids understanding
Simulating an Abstract Machine

C program to produce random bits

```c
#include <stdio.h>
main()
{
    int i, new, fill = 01502;
    for (i = 0; i < 10; i++)
    {
        new = ((fill>>10 & 1)^(fill>>3 & 1));
        fill = (fill << 1) + new;
    }
}

You'll understand this program by next week!

Uses C bitwise operations

  >>  shift right
  <<  shift left
  &  "and" (1 if both 1, 0 otherwise)
  ^  "exclusive or"

Any "general-purpose" machine can be used to simulate any abstract machine. Implications:

  test out new designs
  use old programs
  understand fundamental limitations
Computer Systems and Abstract Machines

Layers of abstraction
understand a machine, then
use it to build a more complex one
substitute new (better) implementations
of abstract machines
develop complex systems by building up

Evolution of "computer"
Box with lights and switches
Box with wires to keyboard and printer
Box with wires to long-term memory (files),
keyboard, and terminal
Bitmapped display with keyboard, mouse,
wires to file system

Programs transform input to output
Files can be input, output, or both

Editor (emacs): create, modify files
Compiler (cc): transform program from text
to machine instructions
Operating System (Unix): invoke programs
Windowing system (X) maintain the illusion of
multiple computer systems
Programs are a sequence of FUNCTIONS

FUNCTIONS are built-in or defined by users

library

user-defined

#include <stdio.h>

float f(float x)
{
    return 2.0 - x*x*x;
}

Functions consist of

a sequence of DECLARATIONS
followed by

a sequence of STATEMENTS

DECLARATIONS name variables, define types

float
    float h;

integer
    int i;

STATEMENTS manipulate data, control execution

assignment
    inc = 0.0;

control
    while (inc < 2.0) { ... }

function call
    printf(...)

Sample program:
print table of values of a function

```c
#include <stdio.h>
float f(float x)
    { return 2.0 - x*x*x; }
main()
    {
        float h;
        h = 0.0;
        while (h < 2.0)
            {
                printf("%4.1f %6.3f\n", h, f(h));
                h = h + 0.1;
            }
    }
```

Your goals
this week: understand programs like this
next week: write programs like this
Contact between your C program
and the outside world

Puts characters on "Standard Output"
default: "terminal" that you're typing at

Formatted output
How do you want the numbers to look?

printf("%4.1f %6.3f\n", i, f(i));

Very flexible: see K&R pp. 13, 154.

Ex: Could print out title line for table with
printf(" x f(x)\n");
Running a program

When you type commands, you are controlling an abstract machine (called the UNIX shell)

COMPILE: convert the program from "human's" language (C) to "machine's" language (stay tuned)

```
$ lcc function.c
```

Result of compilation:
1st try: errors in C program SYNTAX
eventually: a file named a.out

EXECUTE: "start the machine"
starts at machine language instruction corresponding to first statement of main

```
a.out
```

Result of execution:
1st try: errors in C program SEMANTICS
eventually: desired "printf" output
<table>
<thead>
<tr>
<th>x</th>
<th>f(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>2.000</td>
</tr>
<tr>
<td>0.1</td>
<td>1.999</td>
</tr>
<tr>
<td>0.2</td>
<td>1.992</td>
</tr>
<tr>
<td>0.3</td>
<td>1.973</td>
</tr>
<tr>
<td>0.4</td>
<td>1.936</td>
</tr>
<tr>
<td>0.5</td>
<td>1.875</td>
</tr>
<tr>
<td>0.6</td>
<td>1.784</td>
</tr>
<tr>
<td>0.7</td>
<td>1.657</td>
</tr>
<tr>
<td>0.8</td>
<td>1.488</td>
</tr>
<tr>
<td>0.9</td>
<td>1.271</td>
</tr>
<tr>
<td>1.0</td>
<td>1.000</td>
</tr>
<tr>
<td>1.1</td>
<td>0.669</td>
</tr>
<tr>
<td>1.2</td>
<td>0.272</td>
</tr>
<tr>
<td>1.3</td>
<td>0.197</td>
</tr>
<tr>
<td>1.4</td>
<td>0.744</td>
</tr>
<tr>
<td>1.5</td>
<td>1.375</td>
</tr>
<tr>
<td>1.6</td>
<td>2.096</td>
</tr>
<tr>
<td>1.7</td>
<td>2.913</td>
</tr>
<tr>
<td>1.8</td>
<td>3.832</td>
</tr>
<tr>
<td>1.9</td>
<td>4.859</td>
</tr>
</tbody>
</table>

% lcc function.c
% a.out
Characteristic C shortcuts

These two programs are equivalent:

```c
#include <stdio.h>
float f(float x)
    { return 2.0 - x*x*x; }
main()
    {
        float i;
        i = -1.0;
        while (i < 2.0)
            {
                printf("%4.1f %6.3f\n", i, f(i));
                i = i + 0.1;
            }
    }
```

```c
#include <stdio.h>
float f(float x)
    { return 2.0 - x*x*x; }
main()
    {
        float i;
        for (i = -1.0; i < 2.0; i += .1)
            printf("%4.1f %6.3f\n", i, f(i));
    }
```

Concise programs are the norm in C
Many other languages are more verbose
Example Program: Random Integers

Print 10 random integers
  library function rand (in stdlib.h)
  returns positive integers < RAND_MAX
  RAND_MAX is usually 32768
#include <stdio.h>
#include <stdlib.h>
main()
{
  int i;
  for (i = 0; i < 10; i++)
    printf("%d\n", rand());
}

Output:
  16838
  5758
  10113
  17515
  31051
  5627
  23010
  7419
  16212
  4086
Print 10 random numbers between 0 and 1

```c
#include <stdio.h>
#include <stdlib.h>
main()
{
    int i;
    for (i = 0; i < 10; i++)
        printf("%f
", 1.0*rand()/RAND_MAX);
}
```

C has conventions for converting types in expressions with mixed types

Output:

```
0.513871
0.175726
0.308634
0.534532
0.947630
0.171728
0.702231
0.226417
0.494766
0.124699
```
Example: Print 9-by-9 random pattern

```c
#include <stdio.h>
#include <math.h>
main()
{
    int i, j;
    for (j = 0; j < 9; j++)
    {
        for (i = 0; i < 9; i++)
        {
            if ((rand()>>13)&1)
                printf("*");
            else printf(" ");
        }
        printf("\n");
    }
}
```

Q: Why not just use the following test?
   if (rand() % 2) ...
A: Random numbers are not random

Ex:
   often, rightmost bits alternate
   depends on implementation
   (see next slide)

Never can have *all* properties of random bits
Ex: sequence is always the same!

Moral: check assumptions about library functions
Another Example: Gambler's Ruin

Simulate gambler placing $1 even bets
How long does the game last?

```c
#include <stdio.h>
#include <stdlib.h>
main()
{
    int i, cash, seed;
    scanf("%d %d", &cash, &seed);
    srand(seed);
    while (cash > 0)
    {
        if ((rand()>>13)&1) cash++; else cash--;
        for (i = 0; i < cash; i++) printf(" ");
        printf("*
");
    }
}
```

`scanf` function takes input from terminal
`srand` initializes random number generator

. % a.out % a.out
. 4 1231 4 1234
.    *    *
.    *    *
.    *    *
.    *    *
.    *    *
.    *    *
.    *    *
.    *    *
.    *    *
.    *    *
.    *    *
.    *    *
.    *    *
Hmmm.
#include <stdio.h>
#include <stdlib.h>
int doit(int cash)
{
    int cnt;
    for (cnt = 0; cash != 0; cnt++)
        if (((rand()>>13)&1) cash++ else cash--;
    return cnt;
}
main()
{
    int j, t;
    for ( j = 4; j < 10; j++)
    {
        printf("%2d  ", j);
        for (t = 0; t < 5; t++)
            printf(" %7d", doit(j));
        printf("\n");
    }
}

Output:
2  2  6  304  2  2
3  33  17  15  53  29
4  22  1024  7820  22  54
5  243  25  41  7  249
6  494  14  124  152  14
7  299  33  531  49  93
8  218  10650  36  42048  248
9  174090315  83579  299  759  69

Well-studied random process in mathematics
Ref: Feller. Probability Theory and Applications

LEVELS OF ABSTRACTION
random bit, ruin sequence, experiment
User Interface
Programming Environment
Operating System

Layers of abstraction in Unix

bare hardware
machine language
kernel
User level (C programming)
Command level (shell)
Window system

standard "programmer's" interface
“Everything in UNIX is a file”

Abstract mechanism for storage

file:
    sequence of bytes

directory:
    “/”
    sequence of files (and directories)

filename:
    sequence of directory names
    on the path from “/” to the file
File manipulation commands

cat, more
  show the contents

cp
  copy

rm
  remove (delete)

mv
  move (rename)

ls
  list file names

mkdir, rmdir
  create, delete directory

pwd
  name of current directory

cd
  change directory

  ..
  current directory
  ...
  parent directory
  ~
  my home directory
  ~xx
  xx's home directory

chmod
  change permissions mode

  *
  any sequence of characters
DONT TYPE "rm *"
Shell

Command interface to UNIX

Just another programming language
sequences of instructions
mv file1 tmp; mv file2 file1; mv tmp file2
variables
printenv
arguments, flags
ls -lt *.c
conditional
looping
...

"EXTENSIBLE"
add a new command with
cc mycommand.c
mv a.out mycommand
also can add new commands with
chmod 755 doit
doit
where "doit" is a file with shell commands

Primary use
low-overhead "programming" to
manipulate files
invoke commands
Some Unix commands

- lpr output to printer
- man, apropos online documentation
- grep, awk, sed pattern search (stay tuned)
- sort sort the lines
- diff show differences
- cal, date, time time utilities
- mail, news, pine communication
- bc, dc calculators
- cc, lcc, gcc C compile
- ed, vi, troff text in and out (old)
- history past commands typed

Over 2500 "standard" commands
Thousands more "available" programs

- emacs, tex, latex text in and out
- netscape access web
Filters and pipes

Standard Input, Standard Output

abstract files for command interfaces

Redirection:
  standard input from file
  standard output to file

  a.out > saveanswer
  sort < myfile > myfilesorted

Piping:
  connect standard output of one command
to standard input of the next

  ls | wc -l
  plotprog | lpr
  gamblerall | avg

Don’t confuse redirection and piping

  plotprog > lpr
Multiprocessing

Abstraction provided by operating system
multiple "virtual" machines for your use
outgrowth of 1960s "time-sharing"
not found on 1st-generation PC OS's

Multiple windows "active"??

Ex:
emacs hello.c &

ampersand indicates "do this in the background"
alternatively, could use ctrl-z

% emacs hello.c &
[1] 18439
% netscape &
[2] 18434
% jobs
[1] + Running emacs hello.c
[2] - Running netscape
%

For COS126
one window for editor
one window for UNIX commands
lcc, a.out, ls, cp
[one window for output]
Operating systems

Multics
  timesharing
  file system, protection
  virtual machines

OS/360

Macintosh
  windows, mouse

DOS
  PC standard OS

NextStep
  graphical interface
  (programmers use UNIX)
  future Mac OS?

UNIX
  C language, bootstrapped implementation
  integrated command structure
  simplified, integrated file system
  used by most programmers

Windows 95
An imaginary machine, similar to
  * ancient early computers
  * today's microprocessors

Box with switches and lights, maybe TTY

Use to introduce
  * machine language programming
    (how C program relates to machine)
  * computer architecture
    (how machine works)

Inside the box
  CPU (central processing unit)
  256 16-bit words of memory
  8 registers
  PC (program counter)
Data Representations

Machine consists of two-state ("ON-OFF") switches and lights
Use binary encoding to represent values
Ex:

\[ .6375 = 0001100011100111 \]

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1</td>
</tr>
<tr>
<td>12 11</td>
</tr>
<tr>
<td>2 2</td>
</tr>
</tbody>
</table>

\[ .6375 = 4096 + 2048 + 128 + 64 + 32 + 4 + 2 + 1 \]

Hexadecimal (base-16) notation provides shorthand binary code four bits at a time

<table>
<thead>
<tr>
<th>0000 0001 0010 0011 0100 0101 0110 0111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1000 1001 1010 1011 1100 1101 1110 1111</td>
</tr>
<tr>
<td>8 9 A B C D E F</td>
</tr>
</tbody>
</table>

Ex:

\[ .6375 = 0001100011100111 \]

<table>
<thead>
<tr>
<th>1 8 E 7</th>
</tr>
</thead>
</table>

\[ .6375 = 1*16^3 + 8*16^2 + 14*16^1 + 7*16^0 \]

\[ = 4096 + 2048 + 224 + 7 \]

EVERYTHING is encoded in binary (hex) integers machine instructions, text, reals, ...
TOY machine memory

Contents of machine in hexadecimal ("dump")

PC: 0010

R0: R1: R2: R3: R4: R5: R6: R7:
0000 0788 B700 0010 0401 0002 0003 00A0

00: 0000 0000 0000 0000 0000 0000 0000 0000
08: 0000 0000 0000 0000 0000 0000 0000 0000
10: 9222 9120 1121 A120 1121 A121 7211 0000
18: 0000 0001 0002 0003 0004 0005 0006 0007
20: 0008 0009 00A 00B 00C 00D 00E 00F
28: 0000 0000 0000 FE10 FACE CAFE ACED CEDE

Programmers still look at dumps, even in 1990s

Contents of memory
record of what program has done
determines (with PC) what machine will do
Program: sequence of instructions

Instruction:
  16-bit word (interpreted one way)

Data:
  16-bit word (interpreted other ways)

Sixteen different instructions in TOY
  0: halt
  1: add
  2: subtract
  3: multiply
  4: system call
  5: jump
  6: jump if greater
  7: jump and count
  8: jump and link
  9: load
  A: store
  B: load address
  C: xor
  D: and
  E: shift right
  F: shift left
PC (program counter) is memory address of "current" instruction

Instructions change contents of registers and memory (and PC) in specific, well-defined ways

To run a program
* load the program and data (set switches, press LOAD for each word)
* set switches to address of first instruction
* press GO

GO button
* loads PC from address switches
* initiates FETCH-INCREMENT-EXECUTE cycle
* machine runs until halt instruction hit

FETCH (get instruction from memory into CPU)
INCREMENT program counter (PC)
EXECUTE (may require data from or to memory)

Output:
read contents of memory word in lights
system call can write output to an output device (tty)
Instruction Formats

FORMAT 1: register-register

4 bits 4 bits 4 bits 4 bits
opcode dest regA regB

Ex: 1234 means
add register R3 and R4
put the result in R2
R2 <- R3 + R4

Other instrs: sub, mult, xor, and

FORMAT 2: register-memory, register-immediate

4 bits 4 bits 8 bits
opcode dest addr/const

Ex: 9234 means “load memory loc 34 (hex) into R2”
R2 <- mem[34]
Ex: A234 means “store R2 into memory loc 34”
mem[34] <- R2
Ex: B234 means “load the value 0034 into R2”
R2 <- 0034

Other instrs: shifts, halt, system call, jumps
Sample TOY program 1

PROGRAM: sequence of instructions in memory
Set PC anywhere, press GO:
what happens is determined solely by
  * contents of memory
  * rules defining instructions

Ex: Suppose memory locations 10-1F contain
  10: B001 B200 B101 1221 1110 1221 1110 1221
  18: 1110 1221 1110 1221 1110 1221 0000 0000

Set PC to 10. Press GO. What happens?
Step-by-step trace:
  10: B001    R0 <- 0001
  11: B200    R2 <- 0000          0000
  12: B101    R1 <- 0001     0001
  13: 1221    R2 <- R2 + R1       0001
  14: 1110    R1 <- R1 + R0  0002
  15: 1221    R2 <- R2 + R1       0003
  16: 1110    R1 <- R1 + R0  0003
  17: 1221    R2 <- R2 + R1       0006
  18: 1110    R1 <- R1 + R0  0004
  19: 1221    R2 <- R2 + R1       000A
  1A: 1110    R1 <- R1 + R0  0005
  1B: 1221    R2 <- R2 + R1       000F
  1C: 1110    R1 <- R1 + R0  0006
  1D: 1221    R2 <- R2 + R1       0015
  1E: 0000    halt

Computes  1 + 2 + 3 + 4 + 5 + 6 = 21
Suppose memory locations 10-17 contain

10: B006 B201 B101 1221 1110 6113 0000 0000

Set PC to 10. Press GO. What happens?

Step-by-step trace:

10: B106    R1 <- 0006       0006
11: B200    R2 <- 0000            0000
12: B001    R0 <- 0001
13: 1221    R2 <- R2 + R1         0006
14: 2110    R1 <- R1 - R0    0005
15: 6113    jump if (R1 > 0)
13: 1221    R2 <- R2 + R1         000B
14: 2110    R1 <- R1 - R0    0004
15: 6113    jump if (R1 > 0)
13: 1221    R2 <- R2 + R1         000F
14: 2110    R1 <- R1 - R0    0003
15: 6113    jump if (R1 > 0)
13: 1221    R2 <- R2 + R1         0012
14: 2110    R1 <- R1 - R0    0002
15: 6113    jump if (R1 > 0)
13: 1221    R2 <- R2 + R1         0014
14: 2110    R1 <- R1 - R0    0001
15: 6113    jump if (R1 > 0)
13: 1221    R2 <- R2 + R1         0015
14: 2110    R1 <- R1 - R0    0000
15: 6113    jump if (R1 > 0)
16: 0000    halt

Computes

\[ N + (N-1) + ... + 3 + 2 + 1 = \frac{N(N+1)}{2} \]

for *any* value N loaded into R1
Horner's method

Problem:

evaluate \(2x^3 + 3x^2 + 9x + 7\) at \(x = 10\)

assume "data" stored in locations 30--34

\[
\begin{array}{cccccc}
  x & a & b & c & d \\
  30: & 000A & 0002 & 0003 & 0009 & 0007 \\
  31: & 0000 & 0000 & 0000 & 0000 \\
\end{array}
\]

First try:

compute \(x^3\), mult. by \(a\); compute \(x^2\), ...
(cumbersome, inefficient)

Efficient algorithm (Horner's method):

rewrite \(ax^3+bx^2+cx+d\) as \(((ax+b)x+c)x+d\)

10: 9430 R4 <- M[30] 000A x
11: 9531 R5 <- M[31] 0002 a
12: 3554 R5 <- R5 * R4 0014 a*x
14: 1556 R5 <- R5 + R6 0017 a*x+b
15: 3554 R5 <- R5 * R4 00DC (a*x+b)*x
16: 9633 R6 <- M[33] 0009 c
17: 1556 R5 <- R5 + R6 00E5 (a*x+b)*x + c
18: 3554 R5 <- R5 * R4 0956 ((a*x+b)*x+c)*x
19: 9634 R6 <- M[34] 0007 d
1A: 1556 R5 <- R5 + R6 095D ((a*x+b)*x+c*x)+d
1B: 4502 write R5 to tty

Converts from decimal to hexadecimal

2397 in base 10 equals 95D in hexadecimal

Does polynomial evaluation for arbitrary \(x\)
* many applications
* one raison d'être for early machines
Basic Characteristics of TOY machine

"von Neumann" machine
* instructions and data in same memory
* can change program (control) w/o rewiring
  * immediate applications
  * profound implications

sufficient power to perform any computation
(subject only to memory constraints)

similar to real machines
  helps focus attention on costs
  Ex: Horner's algorithm

simple high-level control

Abstract machine
  implement using any technology
  change implementation *without* having
    to change programs
  simulate on another machine
Real machines

Chip on a board

Input and Output
  to and from chip
  to and from machine
  to and from external devices

Much more memory
  * billions of bits, not 4096
  * 32 (or 64) bits per word, not 16

More kinds of instructions
More kinds of built-in data formats
  * large integers
  * floating point
  * character strings

More ways to address memory

SAME BASIC ABSTRACTIONS
Box with switches and lights
   CPU (central processing unit)
   256 16-bit words of memory
   8 registers (1st eight words of memory)

Hexadecimal
   shorthand notation for bits

to LOAD a memory word
   set address switches
   set memory switches
   press LOAD

Program: sequence of instructions

to RUN a program
   set address switches (to initialize PC)
   press GO (or STEP for just one instr.)
to LOOK at a memory word
   set address switches
   press LOOK
   examine lights

Same basic principles as
   early computers
   modern microprocessors
TOY instructions

Instruction formats

FORMAT 1: opcode, r0, r1, and r2
FORMAT 2: opcode, r0, and 8-bit addr

TRANSFER between registers and memory

9: load          r0 <- mem[addr]
A: store         mem[addr] -> r0
B: load address  r0 <- addr

ARITHMETIC operations

1: add           r0 <- r1 + r2
2: subtract      r0 <- r1 - r2
3: multiply      r0 <- r1 * r2

LOGICAL operations

C: xor           r0 <- r1 ^ r2
D: and           r0 <- r1 & r2
E: shift right   r0 <- r0 >> addr
F: shift left    r0 <- r0 << addr

CONTROL

0: halt          halt
4: system call   print r0 on tty
5: jump          pc <- addr
6: jump if positive if (r0 > 0) pc <- addr
7: jump and count r0 <- r0 - 1
                     if (r0 != 0) pc <- addr
8: jump and link  r0 <- pc; pc <- addr
Version 6.0 of the Toy architecture is incompatible with previous versions. Among other things, we have changed the way addressing is done and we have bagged the divide instruction.

All software written for old machines must be reimplemented. People who took CS126 last year learned about some obsolete version.

Don't tell the customers.
Indexed Addressing

Load, store
  addresses hardwired into instructions
  flexibility needed for memory access

Solution:
  use CONTENTS of register as address

How to specify instruction?
  * eight registers
  * four bits to specify register
  * only *need* three

For all Format 1 instructions
  define leading bit of second digit
  to be the INDEX bit
   if it's 1, add r1 and r2 to get addr
   if it's 0, take address as before

Ex:
  if R2 is 0030 and R4 = 0002
  then 9F24 loads mem[32] into r7

Note: B923 is equivalent to 1123
  (these things happen in machine design!)
Put Fibonacci numbers
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
into memory, starting at location 30
(memory use corresponds to C array)

10: B10E    R1 <- 000E
11: B001    R0 <- 0001
12: B230    R2 <- 0030     a
13: A030    mem[30] <- 1    a[0] = 1
14: A031    mem[31] <- 1    a[1] = 1
15: B300    R3 <- 0         i = 0
16: B401    R4 <- 1         j = 1
17: B502    R5 <- 2         k = 2
18: 9E23    R6 <- mem[R2 + R3]  a[i]
19: 9F24    R7 <- mem[R2 + R4]  a[j]
1A: 1667    R6 <- R6 + R7
1B: AE25    mem[R2 + R5] <- R6  a[k]
1C: 1330    R3++         i++
1D: 1440    R4++         j++
1E: 1550    R5++         k++
1F: 7118    to 18 if --R1 > 0

Result of pressing GO with PC at 10:
0010: B10E B001 B230 A030 A031 B300 B401 B502
0018: 9E23 9F24 1667 AE25 1330 1440 1550 7118
0030: 0001 0001 0002 0003 0005 0008 000D 0015
0038: 0022 0037 0059 0090 00E9 0179 0262 03DB

Food for thought:
what happens if we change mem[12] to B219??
Press GO, computer either
* executes some instructions and halts
* gets caught in a loop

"infinite loop"
  puzzles and/or panics programmers
  (it will happen to you!)

xx:
  ...
  ...
  ...
  50xx

Often more complicated

Can't know whether or not a program
  will loop, in general

Control structures (for and while) help
  manage branching, avoid looping

Can always stop TOY by pulling the plug!

Profound implications (stay tuned)
Logical Instructions

C: xor
D: and
E: shift right
F: shift left

xor, and: bit-by-bit operations
shift: move bits

MASKing with "and instruction"

1010010x01111010
0000000100000000
00000000x00000000

Can implement other logical operations

<table>
<thead>
<tr>
<th>a b</th>
<th>&amp;</th>
<th>^</th>
<th>(a &amp; b)</th>
<th>^</th>
<th>(a ^ b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Ex: suppose that memory locations 10-15 contain
10: 911F B000 1210 1310 E203 E30A C323 B401
18: D334 F101 C113 0000 0000 0000 0000 0684
Set PC to 10. Press GO. What happens?

Step-by-step:
10: 911F    R1 <- 0684      0000011010000100
11: B000    R0 <- 0000
12: 1210    R2 <- R1 + R0   0000011010000100
13: 1310    R3 <- R1 + R0   0000011010000100
14: E203    R2 <- R2 >> 3   0000000011010000
15: E30A    R3 <- R3 >> 10  0000000000000001
16: C323    R3 <- R2 ^ R3   00000000011010001
17: B401    R4 <- 0001      0000000000000001
18: D334    R3 <- R3 & R4   0000000000000001
19: F101    R1 <- R1 << 1   0000110100001000
1A: C113    R1 <- R1 ^ R3   0000110100001001
1B: 0000    halt

Simulates one step of LFBSR of Lecture 1
To implement functions
  assume parameter values in registers
  assume return value in register
  use indexed jump to return

Ex: function to compute \( a \) to the \( b \)-th power
  \( o \) in R0
  \( a \) in R1
  \( b \) in R2
  \( \text{addr} \) in R4
  \( \text{result} \) in R3

Implementation computes \( a \) to the \( b \)-th power
  by looping \( b \) times
  multiplying \( R3 \) by \( a \) each time

\[
\begin{align*}
20: & \quad B301 \quad \text{R3} \leftarrow 0001 \\
21: & \quad 1223 \quad \text{R2}++ \\
22: & \quad 5024 \quad \text{jump to 24} \\
23: & \quad 3331 \quad \text{R3} \leftarrow \text{R3} \times \text{R1} \\
24: & \quad 7223 \quad \text{loop to 23 if } --\text{R2} > 0 \\
25: & \quad 5804 \quad \text{jump to \text{addr} in R4}
\end{align*}
\]

CALLING convention
  agreement between function and
  calling program on parameters and
  return address
Adhere to calling conventions to get function to perform computation with different parameter values

Ex: program that calls the function on the previous slide twice to compute $x^4 + y^5$

- $x$ in mem loc 1E
- $y$ in mem loc 1F

10: B000    R0 ← 0
11: 911E    R1 ← $x$
12: B204    R2 ← 4
13: 8420    R3 ← $x^4$ (using function)
14: 1530    R5 ← R3
15: 911F    R1 ← $y$
16: B205    R2 ← 5
17: 8420    R3 ← $y^5$ (using function)
18: 1535    R5 ← $x^4 + y^5$

Precious resource: registers

Call a function from within a function? (stay tuned)
Representations

We've been assuming nonnegative integers

negative integers: use "two's complement"

\[
\begin{align*}
0000000000000100 & \quad 4 \\
0000000000000011 & \quad 3 \\
0000000000000010 & \quad 2 \\
0000000000000001 & \quad 1 \\
0000000000000000 & \quad 0 \\
1111111111111111 & \quad -1 \\
1111111111111110 & \quad -2 \\
1111111111111101 & \quad -3 \\
1111111111111100 & \quad -4
\end{align*}
\]

big integers: could use "multiple precision"

multiple words per integer
required for multiply, divide?

real numbers: could use "floating point"
like scientific notation

character strings: could use ASCII code
8 bits/character (packed/unpacked)

Details (for real machines): stay tuned

Does a C program know the representation?

ideally: NO
in practice: YES
TOY perspective

MACHINE LANGUAGE:
  hex codes we've used
ASSEMBLY LANGUAGE:
  names for opcodes, regs, memory locs
  1-1 translation to machine language

TOY has the same basic computational tools that we use in C
  data representations
    integer, float, char
  control
    assignment, loop, conditional functions

“All” programming languages have these fundamental characteristics

COMPILER translates a program from high-level language like C to low-level machine language like TOY

Different ways to express same underlying computation (algorithm)

Contents of memory
  DATA if referenced by instruction
  INSTRUCTION if referenced by PC

Essential characteristic of VON NEUMANN MACHINE
TOY machine now available!

Put a machine language program in a file
4-digit hex numbers to be loaded at 10
B102
1111
A120
0000

mail it to us

Course staff will
load your program
put 10 in the address switches
press GO
mail back a dump of memory

Easier way:
cp ~cs126/toy/toy.c .
cold toy.c -o TOY
TOY < myprog.toy

Example programs also available
TOY < ~cs126/toy/horner.toy
#include <stdio.h>
int R[8], mem[256]; pc = 16;
main()
{
    int i, inst, op, addr, r0, r1, r2;
    for (i = 0; i < 256; i++) mem[i] = 0;
    for (i = 16; i < 256; i++)
        if (scanf("%X", &mem[i]) == EOF) break;
    do
    {
        inst = mem[pc++];
        op = (inst >> 12) & 0XF;
        addr = inst & 0XFF;
        r0 = (inst >> 8) & 0X7;
        r1 = (inst >> 4) & 0X7; r2 = inst & 0X7;
        if (inst & 0X0800) addr = R[r1]+R[r2];
        switch (op)
        {
            case 0: break;
            case 1: R[r0] = R[r1] + R[r2]; break;
            case 2: R[r0] = R[r1] - R[r2]; break;
            case 3: R[r0] = R[r1] * R[r2]; break;
            case 4: printf("%X\n", addr); break;
            case 5: pc = addr; break;
            case 6: if (R[r0]>0) pc = addr; break;
            case 7: if (--R[r0]) pc = addr; break;
            case 8: R[r0] = pc; pc = addr; break;
            case 9: R[r0] = mem[addr]; break;
            case 10: mem[addr] = R[r0]; break;
            case 11: R[r0] = addr; break;
            case 12: R[r0] = R[r1] ^ R[r2]; break;
            case 13: R[r0] = R[r1] & R[r2]; break;
            case 14: R[r0] = R[r0] >> addr; break;
            case 15: R[r0] = R[r0] << addr; break;
        }
    } while (op != 0);
}
TOY dump

Use C arrays for memory, registers

```c
int R[8], mem[256]; pc = 16;
dump()
{
    int i, j;
    printf("pc: %04X\n", pc);
    printf("regs: ");
    for (i = 0; i < 8; i++)
        printf("%04X ", R[i]);
    printf("\n");
    for (i = 0; i < 32; i++)
    {
        printf("\n%04X: ", 8*i);
        for (j = 0; j < 8; j++)
            printf("%04X ", mem[8*i+j]);
    }
    printf("\n");
}
```

Dump is in hex, 8 words/line, 4 digits/word

```
0000: 0000 0004 0000 0000 0000 0000 0000 0000
0008: 910A 110A 0002 0000 0000 0000 0000 0000
```

Easy to add other functions to simulator

- trace
- single-step
- breakpoint debugging

... Simulated more powerful than TOY itself!

[TOY probably cheaper] Want to design a new machine?

- use the simulator!
- [much cheaper than building a machine]
Correspondence between basic C constructs and basic TOY mechanisms

<table>
<thead>
<tr>
<th>C</th>
<th>TOY</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>indexed addressing</td>
</tr>
<tr>
<td>arithmetic expressions</td>
<td>add, mul</td>
</tr>
<tr>
<td>logical expressions</td>
<td>xor, and, shifts</td>
</tr>
<tr>
<td>switch, if, for, while</td>
<td>conditional jumps</td>
</tr>
</tbody>
</table>

Translate TOY program to C program? easy

Translate C program to TOY program? straightforward (if tedious)

Translate SIMULATOR to TOY program? why not??

BOOTSTRAPPING

build "first" machine
implement simulator
modify simulator to try new designs (still going on!)
Data Structure
* mechanism for organizing information
  arrays
  structs
  linked lists
  binary trees
  (compound structures)

Data Type
* set of values
* collection of operations on those values
  int, float, char
  strings
  stacks
  queues

Interface
Implementation
Client

Abstract data type
Object-oriented programming
Array

Fundamental data structure
store values sequentially in memory
associate INDEX with each value
use index to quickly access Nth for any N

Example: symbolic manipulation of polynomials

C representation of $x^{14} + 3x^5 + 12$:
```c
int p[15];
for (i = 0; i < 15; i++) p[i] = 0;
```

TOY representation of $x^{14} + 3x^5 + 12$:
```
22: 000C
23: 0000
24: 0000
25: 0000
26: 0000
27: 0003
28: 0000
29: 0000
2A: 0000
2B: 0000
2C: 0000
2D: 0000
2E: 0000
2F: 0000
30: 0001
```

Advantages:
can get to each item quickly
index carries implicit info, takes no space

Disadvantage: Uses up space for unused items
Linked List

Fundamental data structure
store values anywhere in memory
associate LINK with each value
use link to quickly access the NEXT value

TOY representation of $x^{14} + 3x^5 + 12$:
22: 000E
23: 0001
24: 004D
...
38: 0000
39: 000C
3A: 0000
...
4D: 0005
4E: 0003
4F: 0038

Advantage: space proportional to amount of info
Disadvantage: can only get to *next* item quickly

C representation of $x^{14} + 3x^5 + 12$: ???

Need to know:
how to associate pieces of information
how to specify links
how to reserve memory to be used
how to use links to access information
To associate pieces of info, use C structures

typedef struct { int p; int q; } Rational;
float x; Rational t;
x = (1.0)*t.p/t.q

To specify links, use "*" (pointers)
typedef struct node* link;
struct node
    { int coef; int exp; link next; }; link p;

To reserve memory for a structure, use "malloc" (library routine declared in stdlib.h)
p = malloc(sizeof *p);

To use a pointer to access information, use "->"
p->coef = 1;

Build the list for $x^{14} + 3x^5 + 12$
p = malloc(sizeof *p)
p->coef = 1; p->exp = 14;
q = malloc(sizeof *p)
q->coef = 3; q->exp = 5;
r = malloc(sizeof *p)
r->coef = 12; r->exp = 0;
p->next = q; q->next = r; r->next = NULL;

STUDY THIS: Tip of the iceberg!
Digression: pointers

POINTERS (addresses)
Used to implement BOTH arrays and linked lists

ARRAY
. \( p \) pointer to array (first item)
. \( *p \) first array item
. \( p+1 \) pointer to second array item
. \( *(p+1) \) second array item
.
. \( *(p+i) \) \((i+1)st\) array item
. \( p[i] \) alternate way to say the same thing

LINKED LIST
. \( p \) pointer to structure
. \( *p \) structure
. \( (*p).coef \) field in structure
. \( p->coef \) alternate way to say the same thing
Stacks, Queues

Prototypical data types
data structures
defined by operations

STACK ("last in, first out")
push: add info to the data structure
pop: remove the info most recently added
(initialize, test if empty)

QUEUE ("first in, first out")
put: add info to the data structure
get: remove the info LEAST recently added
(initialize, test if empty)

Separate implementation from specification

INTERFACE: specify the allowed operations
IMPLEMENTATION: provide code for ops
CLIENT: code that uses them

Could use either
array or linked list
to implement either
stack or queue

Client can work at higher level of abstraction
(stay tuned)
Stack interface and client

Interface (STACK.h)

void STACKinit();
int STACKempty();
void STACKpush(int);
int STACKpop();

Client (myprog.c)

#include "STACK.h"
main()
{
  int a, b;
  ...
  STACKinit(N);
  ...
  STACKpush(a);
  ...
  b = STACKpop();
  ...
}

Client uses data type, without regard to how it is implemented
Array implementation of stack (of ints)

Push and pop at the end of the array

#include <stdlib.h>
#include "STACK.h"
int s[1000];
int N;

void STACKinit()
{
    N = 0;
}

int STACKempty()
{
    return N == 0;
}

void STACKpush(int item)
{
    s[N++] = item;
}

int STACKpop()
{
    return s[--N];
}

Client and implementation both include STACK.h
(could be compiled separately)
can be compiled with one command
cc myprog.c stackArray.c
a.out

Problem: have to reserve space for max size
# Linked-list implementation of stack

Push and pop nodes at the front of the list

```c
#include <stdlib.h>
typedef struct STACKnode* link;
struct STACKnode { int item; link next; };
static link head;
link NEW(int item, link next)
    { link x = malloc(sizeof *x);
        x->item = item; x->next = next;
        return x;
    }
void STACKinit(int maxN)
    { head = NULL; }
int STACKempty()
    { return head == NULL; }
STACKpush(int item)
    { head = NEW(item, head); }
int STACKpop()
    { int item = head->item;
        link t = head->next;
        free(head); head = t;
        return item;
    }
```

Switch implementations without changing interface or client
```
cc myprog.c stackList.c
a.out
```
Polish notation

Practical example of use of stack abstraction

Put operator after operands in expression
Use stack to evaluate
 operand: push it on stack
 operator: pop operands, push result
Systematic way to save intermediate results

Ex: convert 97531 from hex to decimal

\[
\begin{array}{cccccccccc}
9 & 16 & 16 & 16 & 16 & * & * & * & * & 7 \\
5 & 16 & 16 & * & * & 3 & 16 & 16 & * & 1 \\
& 9 & 16 & \ldots & 589824 & 7 & 16 & 589824 & 28672 & 1280 & 48 & 1 \\
& 589824 & 28672 & 1280 & 49 & 589824 & 28672 & 1329 & 589824 & 30001 & 619825 \\
\end{array}
\]

Note: HORNER's METHOD (see lecture 4)

\[
9 \times 16 + 16 \times 5 + 16 \times 3 + 16 \times 1 + 1
\]
Stack never has more than two numbers on it!
Language is postfix
Abstract stack machine

Ex: convert 97531 from hex to decimal
   9 16 mul 7 add 16 mul 5 add
   16 mul 3 add 16 mul 1 add

Stack:
   operands for operators
   arguments for functions
   return values for functions

Coordinate system:  rotate, translate, scale, ...
Turtle commands:  moveto, lineto, rmoveto, rlineto,
Graphics commands:  stroke, fill, ...
Arithmetic: add, sub, mul, div, ...
Stack commands: exch, dup, currentpoint, ...
Control constructs: if, ifelse, while, for, ...
Define functions:  /XX { ... } def

Everyone’s first program: draw a box
   50 50 translate
   0 0 moveto 0 512 rlineto
   512 0 rlineto 0 -512 rlineto -512 0 rlineto
   stroke
   showpage
Abstract data types (ADT)

A data type that is accessed *only* through the operations in the interface

Example: STACK implementations

Rational data type (Assignment 3) is NOT an ADT
representation is in interface
(possible to fix: see CS217)

Representation is hidden in the implementation
Client program does not need to know
*how* the implementation works

Advantages
* different clients can use the same ADT
* can change ADT without changing clients

Convenient way to organize large programs
decompose into smaller problems
substitute alternate solutions
separate compilation
build libraries

Powerful mechanism
for building layers of abstraction
Q: "Dad, can we play 'war'?"
A: "OK, but we'll stop if it takes too long..."

Q: "How long does it take?"
A: "???

#include <stdio.h>
#include <stdlib.h>

typedef struct cardlist* link;
struct cardlist { int card; link next; };  
link Atop, Abot, Btop, Bbot;
int cnt = 0;

showhand(link);
link shuffle(int);
deal();
play();
main()
{
    play(shuffle(52));
    if (Btop == NULL)
        printf("A wins in %d steps ", cnt);
    if (Atop == NULL)
        printf("B wins in %d steps ", cnt);
}
Represent the cards

Use integers 0-51 for the cards

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
<th>H</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13</td>
<td>26</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>27</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>28</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>29</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>30</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>31</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>19</td>
<td>32</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>33</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>21</td>
<td>34</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>22</td>
<td>35</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>23</td>
<td>36</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>24</td>
<td>37</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>25</td>
<td>38</td>
<td>51</td>
<td></td>
</tr>
</tbody>
</table>

Use linked lists for the hands

typedef struct cardlist* link;
struct cardlist { int card; link next; };  
link Atop, Abot, Btop, Bbot;

War if ((Atop->card)%13) == ((Btop->card)%13)
Sample game of War

A: 10 1 13 0 24 . . . 11 7 29 26 41
B: 51 21 43 38 44 . . . 45 2 50 48 9

A: 1 13 0 24 27 . . . 7 29 26 41
B: 21 43 38 44 6 . . . 2 50 48 9 51 10

A: 13 0 24 27 13 . . . 29 26 41
B: 43 38 44 6 39 . . . 50 48 9 51 10 1 21

A: 0 24 27 13 36 . . . 26 41
B: 38 44 6 39 4 . . . 48 9 51 10 1 21 16 43

A: 24 27 13 36 40 . . . 41
B: 44 6 39 4 5 . . . 9 51 10 1 21 16 43 38 0

A: 27 13 36 40 14 . . . 44 24
B: 6 39 4 5 47 . . . 51 10 1 21 16 43 38 0

A: 13 36 40 14 35 . . . 24
B: 39 4 5 47 12 . . . 10 1 21 16 43 38 0 27 6

A: . . . 24
B: . . . 10 1 21 16 43 38 0 27 6 13 36 40 14 35 39 4 5 47 12
Deal the cards

Function with a linked list as argument
Makes two new linked lists for players A and B
Sets global variables
    Atop, Abot: links to first, last nodes of A
    Btop, Bbot: links to first, last nodes of B
Does *not* create any new nodes

deal(link d)
{
    Atop = d; Abot = d; d = d->next;
    Btop = d; Bbot = d; d = d->next;
    while (d != NULL)
    {
        Abot->next = d; Abot = d; d = d->next;
        if (d == NULL) break;
        Bbot->next = d; Bbot = d; d = d->next;
    }
}

Create and shuffle the deck (algorithm)

Fill an array with integers in order
Make a pass through to shuffle
* pick up a new card
* pick a random position among cards in hand
* exchange new card with card at that position

Ex: 10 cards
- 0 1 2 3 4 5 6 7 8 9
- 0
- 0 1
- 2 1 0
- 2 1 3 0
- 2 4 3 0 1
- 5 4 3 0 1 2
- 5 4 3 0 6 2 1
- 5 4 7 0 6 2 1 3
- 5 4 7 0 6 2 8 3 1
- 5 4 7 0 6 2 8 3 9 1

Pass through array to build list
Create and shuffle the deck (code)

Dynamically allocate array of size N

```c
int a[52]
```
(in case we want to use bigger decks)

see Sedgewick, p. 85

```c
int randI(int i)
{
    return rand() / (RAND_MAX/i + 1);
}
```

```c
link shuffle(int N)
{
    int j, k, t;
    int *a = malloc(N*sizeof(int));
    link x, deck = malloc(sizeof *deck);
    for (k = 0; k < N; k++) a[k] = k;
    for (k = 1; k < N; k++)
    {
        j = randI(k);
        t = a[k]; a[k] = a[j]; a[j] = t;
    }
    x = deck; x->card = a[0];
    for (k = 1; k < N; k++)
    {
        x->next = malloc(sizeof *x);
        x = x->next; x->card = a[k];
    }
    x->next = NULL;
    return deck;
}
```
Starting point for implementation
(“Why do we have wars, anyway?”)

```c
int play(link deck)
{
    int Aval, Bval; link Ttop, Tbot;
    deal(deck);
    while ((Atop != NULL) && (Btop != NULL))
    {
        cnt++;
        Aval = Atop->card % 13;
        Bval = Btop->card % 13;
        Ttop = Atop; Tbot = Btop;
        Atop = Atop->next; Btop = Btop->next;
        Ttop->next = Tbot; Tbot->next = NULL;
        if (Aval > Bval)
        {
            if (Atop == NULL) Atop = Ttop;
            else Abot->next = Ttop;
            Abot = Tbot;
        }
        else
        {
            if (Btop == NULL) Btop = Ttop;
            else Bbot->next = Ttop;
            Bbot = Tbot;
        }
    }
}

lcc peace.c
a.out
```

Game “never” ends, for many (almost all?) deals
(“Maybe *that’s* why we have wars”)

8.7
Add the code for war

Add the following code before the
if (Aval > Bval)
test in "peace" code

while (Aval == Bval)
{
    for (i = 1; i <= 4; i++)
    {
        if (Atop == NULL) return;
        Tbot->next = Atop; Tbot = Atop;
        Atop = Atop->next;
    }
    Aval = (Tbot->card)%13;
    for (i = 1; i <= 4; i++)
    {
        if (Btop == NULL) return;
        Tbot->next = Btop; Tbot = Btop;
        Btop = Btop->next;
    }
    Bval = (Tbot->card)%13;
}
Tbot->next = NULL;

"while" not "if", to handle multiple wars

Game STILL *never* ends:
    thousands of moves, or more

Why?
One bit of uncertainty

Assume two cards in battles are randomly exchanged when picked up

\[
\text{if (randI(2))} \\
\quad \{ \ Ttop = Atop; Tbot = Btop; \ \} \\
\text{else} \ \{ \ Ttop = Btop; Tbot = Atop; \ \}
\]

Typical of simulation applications: proper use of randomness is vital!

Ten typical games

B wins in 60 steps
A wins in 101 steps
B wins in 268 steps
B wins in 218 steps
B wins in 253 steps
A wins in 202 steps
A wins in 229 steps
B wins in 78 steps
B wins in 84 steps
B wins in 656 steps
Problems with simulation

Doesn’t precisely mirror real game

Deal is a “stack”, not a “queue”!

Separate hand, pile
  requires much more code to handle
  example: could have war as pile runs out
  no real reason to simulate that part (?)
  sort-of-shuffle pile after war?

Tradeoff
  convenience for implementation
  fidelity to real game

Such tradeoffs typical in simulation
  try to identify which details matter
“WAR” using abstract data types

Need “first class” queue ADT
  declare variables of type queue
  use them as arguments to functions
  hide representation from clients

Interface, implementation?
  [“object-oriented programming”; stay tuned]

“peace” client code, FYI
  (“QUEUE -> “Q” in identifiers, for brevity)

int play(Q deck)
{
  int Aval, Bval, i; Q T = Qinit(52);
  deal(deck);
  while ( ((!Qempty(A)) && (!Qempty(B))))
  {
    cnt++;
    Aval = Qget(A); Bval = Qget(B);
    if (randI(2))
    {
      Qput(T, Aval); Qput(T, Bval);
    }
    else
    {
      Qput(T, Bval); Qput(T, Aval);
    }
    if (Aval%13 > Bval%13)
    {
      while (!Qempty(T)) Qput(A, Qget(T));
    }
    else
    {
      while (!Qempty(T)) Qput(A, Qget(T));
    }
  }
}

Advantage: avoid details of linked lists
Disadvantages: add details of interface
Q: “So, how long does it take?”
A: “About 10 times through the deck (254 battles)”

Q: “How do you know?”
A: “I played a million games…”

Q: “That sounds like fun!”
A: “Let’s try having bigger battles…”

100000 trials

0  583
1  448
2  337
3  254
4  197
5  155
6  126
7  103
8  87
9  75
Recursive program: one that calls itself

MATHEMATICAL INDUCTION:
To prove $S(N)$
* prove $S(0)$
* prove $S(N)$, assuming $S(k)$ for all $k < N$

Ex:
\[ 0 + 1 + 2 + 3 + \ldots + N = \frac{N(N+1)}{2} \]
* trivially true for $N = 0$
* $0 + 1 + 2 + 3 + \ldots + N$
  \[= 0 + 1 + 2 + \ldots + N-1 + N \]
  \[= \frac{(N-1)N}{2} + N \]
  \[= \frac{N(N+1)}{2} \]

RECURSION:
To compute $f(N)$
* compute $f(0)$
* compute $f(N)$, using $f(k)$ for $k < N$

Ex:
```c
int f(int N)
{
    if (N == 0) return 0;
    return f(N-1) + N;
}
```
Many computations are naturally expressed as recursive programs

ITERATION
another way to write "for" loop

"DIVIDE and CONQUER"
solve a problem by dividing into smaller ones

Ex: root finding

float bisectr(float l, float r)
{
    float m;
    m = (l + r) / 2;
    if ((r - l) < epsilon) return m;
    if (f(m) > 0.0)
        return bisectr(m, r);
    else
        return bisectr(l, m);
}
Digression: binary search

Bisection for integer functions:

```c
int bisectr(int l, int r)
{
    int m;
    m = (l + r) / 2;
    if (r <= l) return m;
    if (f(m) > 0)
        return bisectr(m+1, r);
    else
        return bisectr(l, m-1);
}
```

Suppose an array A has N integers, in order

```
0 1 2 3 4 5 6 7 8 9 10 11 12
1 1 2 5 8 13 21 34 55 89 144 233 377
```

SEARCH PROBLEM: is a given integer v in A?

SOLUTION: use above program with

```c
int f(int k) { return v - a[k]; }
```

Ex: v = 144

```
. l  r  m  f(m)
.  0 12  6  +
.  7 12  9  +
. 10 12 11  -
. 10 10 10  0
```
Simple recursive programs can consume excessive resources

Ex: Compute binomial coefficients

```c
int f(int N, int k)
{
    if ((k < 0) || (k > N)) return 0;
    if (N == 0) return 1;
    return f(N-1, k) + f(N-1, k-1);
}
```

Seems to run for a long time to compute f(20, 10).

Q: Why?
A: Recomputes intermediate results

Simpler example: hard way to compute $2^N$

```c
int f(int N)
{
    if (N == 0) return 1;
    return f(N-1)+f(N-1);
}
```

Takes time proportional to $2^N$ (!)

Solution: DYNAMIC PROGRAMMING

save away intermediate results
see Sedgewick, section 5.3
Implementing recursion

*Any* function call requires
set the values of the parameters
save the "environment"
jump to the first instruction in the function
execute the function
restore the "environment"
continue at the instruction after the call
"return address" (part of environment)

Use pushdown stack for save/restore
  call: push environment
  return: restore environment from stack

Ex:

```
STACK (top at left)

A calls B  B  save(A)
B calls C  C  save(B)  save(A)
C calls D  D  save(C)  save(B)  save(A)
D returns  C  save(B)  save(A)
C calls E  E  save(C)  save(B)  save(A)
E returns  C  save(B)  save(A)
C returns  B  save(A)
B returns  A
```
Traveling Salesman problem

Given a set of points, find the shortest tour connecting all the points

Recursive solution for trying all possibilities

```c
visit(int N)
{
    int i;
    if ( N == 1) { checklength(); return; }  
    for (i = 1; i <= N; i++)
    {
        swap(i, N);
        visit(N-1);
        swap(i, N);
    }
}
```

Takes \( N! \) steps

Can't run for very large \( N \)
no computer can ever run this
to completion for \( N = 100 \) \([10! > 10^{150}]\)

[stay tuned]
Recursion: Dragon Curve

Fold a strip of paper in half n times
      unfold to right angles

   n = 0
   n = 1
   n = 2
   n = 3
   n = 4
   n = 12
Use simplest turtle graphics
F: move forward one step (pen down)
L: turn left
R: turn right

\[ n = 0 \]
\[
F
\]

\[ n = 1 \]
\[
F \ L \ F \]

\[ n = 2 \]
\[
F \ L \ F \ L \ F \ R \ F \]
\[
L \ L \ R
\]

\[ n = 3 \]
\[
F \ L \ F \ L \ F \ R \ F \ L \ F \ L \ F \ R \ F \ R \ F \]
\[
L \ L \ R \ L \ L \ R \ R
\]

\[ n = 4 \]
\[
L \ L \ R \ L \ L \ R \ R \ L \ L \ R \ R \ L \ R \ R \ R
\]
A dragon curve of order $n$ is
* dragon curve of order $n-1$
* $L$
* dragon curve of order $n-1$, backwards

```c
dragon(int n)
{
    if (n == 0) { F(); return; }
    dragon(n-1);
    L();
    nogard(n-1);
}
```

Still need implementation of "nogard"
Backwards dragon curve

To get "nogard" string from "dragon" string
  * reverse the string
  * switch L and R

dragon:  F L F L F R F
reverse: F R F L F L F
nogard:  F L F R F R F

nogard(int n)
{
    if (n == 0) { F(); return; }
    dragon(n-1);
    R();
    R();
    nogard(n-1);
}

Note: easy to make this program nonrecursive
      (replace recursive call with jump to beginning)
Alternate "dragon"

Replace call to "nogard" by nonrecursive version

dragon(int n)
{
    int k;
    if (n == 0) { F(); return; }
    dragon(n-1);
    L();
    for (k = n-2; k >= 0; k--)
    {
        dragon(k);
        R();
    }
    F();
}

F L F L F R F L F L F R F R F

F L F L F R F L F L F R F R F

Points out self-similarities in curve
Nonrecursive dragon curve

\[
\begin{array}{c}
\text{D(3)} \\
\text{D(2)} & \text{N(2)} \\
\text{D(1)} & \text{N(1)} & \text{D(1)} & \text{N(1)} \\
\text{D(0)} & \text{N(0)} & \text{D(0)} & \text{N(0)} & \text{D(0)} & \text{N(0)} \\
\text{F} & \text{L} & \text{F} & \text{L} & \text{F} & \text{R} & \text{F} & \text{L} & \text{F} & \text{L} & \text{F} & \text{R} & \text{F} & \text{R} & \text{F}
\end{array}
\]

To write down the whole dragon curve sequence
* first, put "F" in every other space
* put "L", "R" (alternating) in every other remaining space
* continue until done

\[
\begin{array}{c}
\text{F} & \text{F} & \text{F} & \text{F} & \text{F} & \text{F} & \text{F} & \text{F} \\
\text{F} & \text{L} & \text{F} & \text{F} & \text{R} & \text{F} & \text{F} & \text{L} & \text{F} & \text{L} & \text{F} & \text{R} & \text{F} & \text{R} & \text{F} & \text{R} & \text{F} \\
\text{F} & \text{L} & \text{F} & \text{L} & \text{F} & \text{R} & \text{F} & \text{F} & \text{L} & \text{F} & \text{L} & \text{F} & \text{R} & \text{F} & \text{R} & \text{F} & \text{R} & \text{F} \\
\text{F} & \text{L} & \text{F} & \text{L} & \text{F} & \text{R} & \text{F} & \text{L} & \text{F} & \text{L} & \text{F} & \text{R} & \text{F} & \text{R} & \text{F} & \text{R} & \text{F}
\end{array}
\]

Like Towers of Hanoi (see Sedgewick, section 5.2) requires too much storage (how much?) “ruler function” connects to binary numbers Details? [challenge for the bored]
Class list

192-034-2006 Alam
201-212-1991 Baer
202-123-0087 Bagyenda
177-999-9898 Balestri
232-876-1212 Benjamin
122-999-3434 Berube
...

Desired operations
add student
return name, given ID number

SEARCH KEY

Similar applications
online phone book
airline reservations
"symbol table"
...

GOAL: fast search *and* insert
even for huge databases
Define "Item.h" file to encapsulate item type

typedef int Key;
typedef struct { Key ID; char name[30]; } Item;

#define eq(A, B) ((A.ID) == (B.ID))
#define less(A, B) ((A.ID) < (B.ID))

Keep array of Items, in sorted order
Use bisection method to find Item sought

Item search(int l, int r, Key v)
{
    int m = (l+r)/2;
    if (l > r) return NULLItem;
    if eq(v, key(st[m])) return st[m];
    if (l == r) return NULLItem;
    if less(v, key(st[m]))
        return search(l, m-1, v);
    else return search(m+1, r, v);
}

Ex: search for 25
.  0  1  2  3  4  5  6  7  8  9 10 11 12
.  06 13 14 25 33 43 51 53 64 72 84 97 99
.  06 13 14 25 33 43
.       25 33 43
.        25

[See also Lecture 9; Programs 2.2 and 12.6]
Cost of Binary Search

Q: How many "comparisons" to find a name?
A: log N
   divide list in half each time
   Ex: 5000 -> 2500 -> 1250 -> 625 -> 312 ->
       156 -> 78 -> 39 -> 18 -> 9 -> 4 -> 2 -> 1

log N = number of digits in decimal rep. of N
lg N = number of digits in binary rep. of N

lg(thousand) = 10
lg(million) = 20
lg(billion) = 30

Without binary search, might have to look at everything, so savings is substantial for very large files.

Problem: insert operation is usually slow

Ex: to insert 49

.  0  1  2  3  4  5  6  7  8  9 10 11 12
. 06 13 14 25 33 43 51 53 64 72 84 97 99

have to move larger keys over one position

.  0  1  2  3  4  5  6  7  8  9 10 11 12 13
. 06 13 14 25 33 43 49 51 53 64 72 84 97 99
Linked List Representation

Keep items in a linked list

typedef struct STnode* link;
struct STnode { Item item; link next; }

Advantage of linked representation
  can insert just by changing links
  (no need to "move" anything)

  .  06
  .   13
  .    14
  .     25
  .      33
  .       43
  .        51
  .         49  53
  .           64
  .             72
  .               84
  .                 97
  .                   99

Problem: can't have fast search
ARRAY: fast search, slow insert
LINKED LIST: slow search, fast insert
Binary Search Tree

Use *two* links per node

typedef struct STnode* link;
struct STnode { Item item; link l, r; };

Think of keys printed in order, left to right
take middle name for top, or "root" node
build tree recursively
"l" points to tree for left half
"r" points to tree for right half

.                     51
.         14                      72
.   06          33          53          97
. .    13    25    43    .     64    84    99
.     .  .  .  .  .  .        .  .  .  .  .  .
.   06 13 14 25 33 43 51    53 64 72 84 97 99

NULL links at bottom: "no information here"
Search in Binary Search Trees

Start at "head", link to the root
if current node has key sought, return
go left if key < key in current node
go right if key > key in current node

Item searchR(link h, Key v)
{
    if (h == NULL) return NULLItem;
    if eq(v, key(h->item)) return h->item;
    if less(v, key(h->item))
        return searchR(h->l, v);
    else return searchR(h->r, v);
}

Item STsearch(Key v)
{ return searchR(head, v); }

Search cost:
same as binary search in arrays
*if* tree is balanced

Algorithm works for *any* tree shape
will be slow for some tree shapes

Insert cost?
Insertion in binary search trees

Search for key not in tree
ends on a NULL pointer
node "belongs" there
make a node, link it into the tree

link insertR(link h, Item item)
{ Key v = key(item);
  if (h == NULL)
    return NEW(item, NULL, NULL);
  if less(v, key(h->item))
    h->l = insertR(h->l, item);
  else h->r = insertR(h->r, item);
  return h;
}
void STinsert(Item item)
{ head = insertR(head, item); }
BST construction example
Other types of trees

Need not have precisely two children
Order might not matter

"Family" tree

. me
.
. mom dad
.
. mom's mom mom's dad dad's mom dad's dad

Parse tree ( a * ( b + c ) ) - ( d + e )
.
. -
. *
. +
. a + d e
. b c

UNIX directory hierarchy
.
. /
.
. bin lib etc sys u
.
. bms dlj jma rs sis
.
. book3 C hw java mail src
.
. 0 1 2 3 4
Traversing Binary Trees

Goal: "visit" (process) each node in the tree

visit(link h)
    { printf("%d %s ", h->item.ID, h->item.name);
traverse(link h)
    {
        if (h != NULL)
            {
            traverse(h->l);
            visit(h);
            traverse(h->r);
            }
    }

Goal realized no matter what order
the statements in the "if" are executed

Preorder: visit before recursive calls
Inorder: visit between recursive calls
Postorder: visit after recursive calls

IMPORTANT NOTE:
inorder search provides "free" SORT
in binary search trees!
Preorder traversal with a stack

Visit the top node on the stack
push its children

traverse(link t)
{
    STACKpush(t);
    while (!STACKempty())
    {
        t = STACKpop(); visit(t);
        if (t->r != z) STACKpush(t->r);
        if (t->l != z) STACKpush(t->l);
    }
}

Note: inorder more complicated
(makes one appreciate recursion)

Preorder traversal of tree on slide 5:
.   51 14 06 13 33 25 43 72 53 64 97 84 99
Stack contents:
.
.   14 06 13 33 25 43 72 53 64 97 84 99
.   72 33 33 72 43 72 97 97 99
.   72 72 72

Works for general trees
Generalizes to DEPTH-FIRST SEARCH in graphs
Level order traversal

Use a queue instead of a stack

```c
traverse(struct node *t)
{
    QUEUEput(t);
    while (!QUEUEempty())
    {
        t = QUEUEget(); visit(t);
        if (t->l != z) QUEUEput(t->l);
        if (t->r != z) QUEUEput(t->r);
    }
}
```

Visits nodes in order of distance from root

Level order traversal of tree on slide 5:
. 51 14 72 06 33 53 97 13 25 43 64 84 99
Queue contents:
. 14 72 06 33 53 97 13 25 43 64 84 99
. 72 06 33 53 97 13 25 43 64 84 99
. 33 53 97 13 25 43 64 84 99
. 97 13 25 43 64 84 99
. 43 64 84 99
. 99

Works for general trees
Generalizes to BREADTH-FIRST SEARCH in graphs
Interface between real and abstract machines

Abstract:
  everything in machine represented
  as binary numbers (sequence of 0-1 values)

Real:
  machines constructed of switches, wires
  everything is either "on" or "off"
**Boolean functions of two variables**

**Sixteen different functions**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>constant 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>AND (xy) [decode 00 = 3]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>[decode 10 = 2]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>[decode 01 = 1]</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>XOR (x^y)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>OR (x+y)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NOR (&quot;not or&quot;) [decode 00 = 0]</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>== (&quot;not xor&quot;)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>NOT y (y')</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>NOT x (y')</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>NAND (&quot;not and&quot;)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>constant 1</td>
</tr>
</tbody>
</table>

**AND, OR and NOT (for example) suffice to express *any* boolean function**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x'</th>
<th>x'y</th>
<th>y'</th>
<th>xy'</th>
<th>x'y+xy' = x^y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Boolean functions of three variables

256 different functions

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000001111</td>
<td>x</td>
</tr>
<tr>
<td>0011001111</td>
<td>y</td>
</tr>
<tr>
<td>0101010101</td>
<td>z</td>
</tr>
<tr>
<td>0000000000</td>
<td>constant 0</td>
</tr>
<tr>
<td>0000000001</td>
<td>AND [decode 111 = 7]</td>
</tr>
<tr>
<td>0000010000</td>
<td>[decode 100 = 4]</td>
</tr>
<tr>
<td>0001011111</td>
<td>majority</td>
</tr>
<tr>
<td>011010001</td>
<td>odd parity</td>
</tr>
<tr>
<td>10101010</td>
<td>NOT z</td>
</tr>
<tr>
<td>11001100</td>
<td>NOT y</td>
</tr>
<tr>
<td>11110000</td>
<td>NOT x</td>
</tr>
<tr>
<td>11111110</td>
<td>NAND</td>
</tr>
<tr>
<td>11111111</td>
<td>constant 1</td>
</tr>
</tbody>
</table>

65536 different functions of 4 variables
Sum-of-products

Systematic way to express any function
AND, OR, and NOT

form AND terms for each 1 in the function
use v if it corresponds to v = 1
use v' (NOT v) if it corresponds to v = 0
OR the terms together

Ex: majority function
\[
\begin{align*}
x &: 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \\
y &: 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \\
z &: 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\
m &: 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \\
m &= x'y'z + xy'z + xyz' + xyz
\end{align*}
\]

Example: odd parity function
\[
\begin{align*}
x &: 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \\
y &: 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \\
z &: 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\
p &: 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \\
p &= x'y'z + x'yz' + xy'z' + xyz
\end{align*}
\]
Boolean circuits

Circuits are interconnected "GATES"
Values of variables are on the wires

AND:

OR:

NOT:

Use sum-of-products form of function
Example: majority
\[ m = x'y'z + xy'z + xyz' + xyz \]

Could be done with binary gates
Smaller circuits easy to devise

ANY boolean function can be implemented with AND, OR, and NOT gates
Building gates

Gate implementation is the basis of the interface between real and abstract machines

Switches, relays, vacuum tubes, transistors,...

Transistor: controlled switch

NOT gate (one transistor)

OR gate (three transistors)

AND gate (four transistors)

ANY boolean function can be implemented by wiring together transistors
A useful circuit

DECODER

N "inputs"
2^N "outputs"

Turns on precisely one "output"
address is encoded in "inputs"

\[
\begin{array}{cccc}
  x & y & z \\
  \hline \\
  x'y'z' & 000 \\
  x'y'z & 001 \\
  x'yz' & 010 \\
  x'yz & 011 \\
  xy'z' & 100 \\
  xy'z & 101 \\
  xyz' & 110 \\
  xyz & 111 \\
\end{array}
\]

Converts binary to "unary"
Used to select wires by binary ID (address)
Adder circuit

Goal: add two N-bit numbers x and y
Not a bit-by-bit function of the inputs because of “carry”

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

For each bit
three inputs: (x, y, z)
two outputs: sum, carry

Sum: odd parity circuit
Carry: majority circuit

**ADDER:** string together one copy of circuit for each bit

Timing? [stay tuned]
Adder is a set of switches, wired together
It's not enough to know how to add, we want to remember the answer! (also, we need to know how to count...)

Combinational circuits

characterized by no loops
output well-defined function of input

Sequential circuits

characterized by loops (FEEDBACK)
output depends on TIME as well as input
FLIP-FLOP

Simple sequential circuit

Capable of "storing" a bit
pulse on SET (S) line turns flip-flop on
pulse on RESET (R) line turns flip-flop off

RS flip-flop

"Clocked" flip-flop
extra line to control moment of flip

Clocked RS flip-flop

Use flip-flops to build registers and memory
Register

Sequence of flip-flops for storing bits

LOAD line (and clock)  
controls when new values are loaded

Use registers for
    PC
    memory addresses
    memory buffer
    arithmetic
Memory (bits)

Memory cell
contents on OUT if SELECT and READ on IN used to set/reset if SELECT and WRITE on

Connect SELECT lines to decoder
101 into decoder, SELECT and READ puts value of 101 flip-flop on OUT

Note: bits don’t move; their values propagate
"Bit slice"
build circuit for first bit in every word
copy circuit 16 times for 16-bit word

Long and thin: actual memories split address into
"row" and "column"
Other kinds of flip-flops

"Copy" bit value

D flip-flop

On clock pulse
\[ D = 1 \text{ set} \]
\[ D = 0 \text{ reset} \]

Edge-triggered
make sure bit value isn't used until ready

Master-slave flip-flop
Master-slave flip-flop with feedback

Internal state changes when clock goes on
Output changes when clock goes from on to off

Chain these cells together

|      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| on   | on   |      |      |      |      | 1    | 0    | 0    |      |      |      |      |      |      |      |      |      |      |      |
| off  | on   |      |      |      |      | 0    | 1    | 0    |      |      |      |      |      |      |      |      |      |      |      |
| on   |      |      |      |      |      | 1    | 1    | 0    |      |      |      |      |      |      |      |      |      |      |      |
| off  | off  | on   |      |      |      | 0    | 0    | 1    |      |      |      |      |      |      |      |      |      |      |      |
| on   |      |      |      |      |      | 1    | 0    | 1    |      |      |      |      |      |      |      |      |      |      |      |
| off  | on   |      |      |      |      | 0    | 1    | 1    |      |      |      |      |      |      |      |      |      |      |      |
| on   |      |      |      |      |      | 1    | 1    | 1    |      |      |      |      |      |      |      |      |      |      |      |
| off  | off  | on   |      |      |      | 0    | 0    | 0    |      |      |      |      |      |      |      |      |      |      |      |
High-level view of computer

Memory with feedback, clocked

Each clock tick enables change in memory state

Everything determined by wires, gates EXCEPT clock tick
Levels of abstraction
  raw materials
  transistors
  AND, OR, and NOT gates
  boolean circuits
    Adder
  sequential circuits
    registers, counter
    memory
  control signals
  COMPUTER

physical clock

FETCH-INCREMEN-EXECUTE
Building blocks

Transistor

AND, OR, NOT gates

Adder

Multiplexer (decoder)

Flip-flop

Register

Memory cell

Memory

Counter

CONTROL LINES

Clock
Register Transfer

Wire registers together
output of one to input of the next
one wire per bit

To move information
sequentially turn control lines on, off

Same method moves information
to and from memory
to and from arithmetic circuits
Register Selection

Boolean selection circuit

s selects x
\[ t \text{ selects } y \]

Register selection

S, then LOAD: contents of X to Z
T, then LOAD: contents of Y to Z
Basic Machine Organization

Memory and registers, interconnected
(not all connections shown)

Two types of wires
- data: connect registers (copy signals)
- control: enable connections
Sequences of control signals "move" information

Ex: 20: 9235 (Load R2 with the word at loc 35)

<table>
<thead>
<tr>
<th>step</th>
<th>control signal</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>switch PC--&gt;MAR</td>
<td>PC (20) --&gt; MAR</td>
</tr>
<tr>
<td>1</td>
<td>LOAD MAR</td>
<td>mem[20] --&gt; MRR</td>
</tr>
<tr>
<td>2</td>
<td>main memory R</td>
<td>mem[35] --&gt; MRR</td>
</tr>
<tr>
<td>3</td>
<td>select MRR--&gt;IR</td>
<td>MRR --&gt; IR</td>
</tr>
<tr>
<td>4</td>
<td>LOAD IR</td>
<td>r0 (2) --&gt; RAR</td>
</tr>
<tr>
<td>5</td>
<td>INCPC</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>select IR--&gt;MAR</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>LOAD MAR</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>main memory R</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>select MRR--&gt;RWR</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>LOAD RWR</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>select IR0--&gt;RAR</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>LOAD RAR</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>reg. memory W</td>
<td></td>
</tr>
</tbody>
</table>

Control signal sequence for each instruction

- Load, store: main memory, reg. memory
- Arithmetic insts: reg. memory, ALU
- Jump instructions: PC

MICROPROGRAM: sequence of control signals
Q: How can sequence of control signals be made to happen?
A: Feed counter and op code into decoders

Counter automatically cycles
each tick raises a control signal
opcode selects relevant signal
FETCH-INCREMENT-EXECUTE

Crucial distinctions
clock: externally generated periodic pulse
counter: micro control
  incremented by clock pulse
PC: just a register
  incremented by control pulse
Many details omitted
    most important: slim margin for error
Computer constructed by layering abstractions
    better implementation at low levels
    improves *everything*
Therefore, ongoing search for better switch!

Babbage
    mechanical switches
1940s
    relays, vacuum tubes
1950s
    transistor, core memory
1960s
    integrated circuit
1970s
    microprocessor
1980s
    VLSI
1990s
    integrated systems, parallelism

Future?
    new switch technologies
    quantum computing (not switch-based!)
DNA computing
    [stay tuned!]
Essential and characteristic UNIX tools

- grep
- awk
- sed
- more
- emacs

Learn to use these!

References:

- UNIX Programming Environment
  (Kernighan and Pike)
- AWK Programming Language
  (Aho, Weinberger, and Kernighan)
- "man"

*Not* C programming,
though the tools are (as?/more?) powerful

Useful for variety of practical applications

Directly related to
fundamental tenets of computer science
Context for second half of COS126

Most of us "know"
  how to program in C (but need practice)
  how machine works (but want to know more)

C programming
  deemphasize (specific knowledge in manual)
  reemphasize (context for advanced concepts)
  build efficient programs and systems

Learn to use and exploit new systems
  get small examples working
  go to the documentation
  understand limitations

Appreciate other programming languages
  ex: grep, awk, sh, perl, c++, java, maple
  differ from C (and each other) in "details"
  applications-oriented abstract mechanisms
  cult following

Fundamental limits on computation

Systems programs and programming environments

Applications
General questions

that come up when facing a new problem to solve using a computer:

What primitive objects are important?
Ex. numbers, files, pictures, text, programs
[ Could always do it in C. ]
[ Does another tool allow direct manipulation?]

How long will it take me to do this task?
[ Depends on what tool I use. ]

Have I done something like this before?
[ If so, maybe I should use the same tool. ]
[ Maybe I have some code laying around. ]
[ Does it still work? ]

Will I be doing something like this again?
[ If not, quick hack may be OK. ]

Will I be doing something like this *frequently*?
[ Worthwhile to learn a new tool? ]
[ Worthwhile to *create* a new tool? ]

Has *someone else* done something like this?
[ May be some code laying around. ]

Will *someone else* be doing this in the future?
[ Document the code? ]
[ Make it portable? ]

No easy answers: need to consider alternatives with an *open mind*
grep

general regular expression pattern matching
filter
stdout gets only those lines from stdin
that "match" argument string

Elementary examples:

Does a file contain a string?

grep Smith classlist

Which file contains a string?

grep grep *.sl

Just give me the data of interest...

a.out | grep -v DEBUG
Crossword puzzle or Scrabble too time consuming?

usr/dict/words is list of words in dictionary
25,486 words

Grep and similar tools can be effective in "finding" words

```
grep hh /usr/dict/words
   beachhead
   highhanded
   withheld
   withhold
```

```
grep .a.a.a  /usr/dict/words | wc
   34
```

```
grep .u.u.u /usr/dict/words
   cumulus
```

Ex: Do spell checking by specifying what you know

```
egrep 'n(ie|e|i)ther' /usr/dict/words
```
Name

grep - search file for regular expression

Syntax

grep [option...] expression [file...]

Description

Commands of the grep family search the input files (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output.

Take care when using the characters $ * [ ^ l ( ) and in the expression because they are also meaningful to the Shell. It is safest to enclose the entire expression argument in single quotes ' '.

Options

- c Produces count of matching lines only.
- i Considers upper and lowercase letter identity.
- n Precedes each matching line with its line number.
- v Displays all lines that do not match.

Restrictions

Lines are limited to 256 characters; longer lines are truncated.

See Also

ex(1), sed(1), sh(1)
conventions for grep:

- `c` any non-special char matches itself
- `^` beginning of line
- `$` end of line
- `.` any single character
- `[^...]` any character not in [a-z]
- `r*` zero or more occurrences of `r`

"extended" regular expressions (egrep)

- `(r)`
- `r1 | r2`

specific technical meaning in theoretical CS
[stay tuned, next lecture]
More Examples

Find all references to Java
  grep 'java|Java' text
  grep '[Jj]ava' text

Find all big files
  du -a | egrep "^[1-9][0-9][0-9][0-9]"

Find all lines with dollar amounts on them
  egrep '[$][0-9]+.[0-9]*' myfile
BUG: matches $7A46.
Ex: fix this bug

Matches involving special chars can be complex
Ex: excerpt from "man grep":
  grep -E '\( *[a-zA-Z]*|[0-9]*) *\)' my.txt
This command displays lines in my.txt such as
  ( 783902) or (y), but not (alpha19c).

Note that with grep -E, \( and \) match parens
in the text and ( and ) are special chars that
group parts of the pattern. With grep without
the -E flag, the reverse is true; use ( and )
to match parens and \( and \) to group chars.

Can't "escape" this problem!
Pattern Matching alternatives in UNIX

grep
egrep: extended regular expressions
fgrep: search for multiple patterns
egrep, fgrep are obsolescent
    [try "man grep", "webster obsolescent"]

more: (Try it!)

Substitution (editing), not just matching

emacs, ex (various ways)
    interactive

sed
    filter
    line-by-line editing
sed 's/apples/oranges/g' file

awk: Pattern matching "language"
    matching
    substitution
    pattern manipulation
    variables
    numeric capabilities
    control and logic
Ref:
"man AWK"
The AWK Programming Language
   Aho, Kernighan, and Weinberger

Simple language
   string and data manipulation
   formatting
   selecting

Based on pattern matching

Expressive language, short programs
General-purpose programmable tool
Rapid prototyping

"Automatic"
   input
   field-splitting
   storage management
   declarations, initialization

Running awk:
% awk 'program' inputfile
% command | awk 'program' | command
% awk -f program inputfile
Patterns and Actions

AWK program: sequence of pattern-action pairs

"patterns"
  regular expression
  conditional expression

"actions"
  print
  arithmetic statement
  [full C-like program]

"variables"
  input lines ("records") are divided into "fields"
    delimited by space char (change with -F)
  built-in variables
    NF -- number of fields in current record
    NR -- number of records seen so far
    $0 -- current record
    $i -- ith field in current record
  BEGIN -- matches first line
  END   -- matches last line

No explicit input
Basic AWK cycle
  for every input line
    if pattern matches, perform action
Sample AWK programs

Grep
    awk '/pattern/' file

Line Count
    awk 'END {print NR}' file

Rearrange fields
    awk '{ print $2, $1 }' file

Print long lines
    awk 'length > 72' file

Add up first column, print sum and average
    awk '{ s += $1 } END { print s, s/NR }' file

Compute COS126 midterm grades
    awk 'scores | sort -r +2 | lpr

    {printf("%8s ", $1);}  
    {p = $2+$3+$4+$5; q = $6;}  
    {printf("%4d ", 100*(16*p/40.0 + 20*q/65.0)/36.0);}  
    {printf("%3d%3d%3d%3d%3d ", $2, $3, $4, $5, $6);}'}
Specifying "pattern" for grep can be complex

`^[^aeiou]*a[^aeiou]*e[^aeiou]*i
[^aeiou]*o[^aeiou]*u[^aeiou]*$`

What kinds of patterns can be specified?
match all lines containing an even number?
match all lines containing a prime number?

Which aspects are essential?

**Regular Expression**

- 0 or 1
- (a)
- ab
- a+b
- a*

where a and b are regular expressions

**Ex:**

- (10)*
- (0+011+101+110)*
- (01*01*01*)*
Formal Languages

Language:
a set of strings over a finite alphabet

Every regular expression (RE) describes a language
(the set of all strings that "match")

Regular Language:
any language representable by an RE

What languages are regular?

Examples (all but one of the following are regular)

all bit strings
that begin with 0 and end with 1
whose number of 0’s is a multiple of 5
with more 1’s than 0’s
with no consecutive 1’s

Can cast any computation as a language problem

Start by trying to understand simple languages
Finite State Automata

Simple machine with N states
start in state 0
read a bit
move to new state
depends on bit, current state
stop when last bit read
ACCEPT if in specified state X
REJECT otherwise

Also say FSA "recognizes" input string or "decides" whether input string is in the language it can recognize

Ex:
(10)*

10101010?
013131313
Ex: odd number of 0's

0001110?
01011110

FSAs and REs are equivalent
[stay tuned]
An application

"Bounce" filter:
remove isolated 0's and 1's in a bitstream
input:
0 1 0 0 0 1 1 0 1 1 1 1
output (one-bit delay)
0* 0 0 0 0 0 1 1 1 1 1 1 1

state | in | out
---|---|---
0 | 0 | 0
0 | 1 | 0
1 | 0 | 0
1 | 1 | 1
2 | 0 | 1
2 | 1 | 1
3 | 0 | 0
3 | 1 | 1

State interpretations
0: at least two consecutive 0's
1: seq. of 0's followed by a 1
2: at least two consecutive 1's
3: seq. of 1's followed by a 0
C program to Simulate FSAs

#include <stdio.h>

main(int argc, char*argv[]) {

    int zero[100], one[100]; char c;
    FILE *fsa = fopen(*++argv, "r");
    int state, N;
    for (N = 0; !feof(fsa); N++)
        fscanf(fsa, "%d  %d ", &zero[N], &one[N]);
    state = 0;
    while ((c = getchar()) != EOF)
        if (c == '0') state = zero[state];
             else state = one[state];
        if (state == 0) printf("Accepted ");
             else printf("Rejected ");
}

Ex: FSA to decide if input is divisible by 3

[magic!?]

S 0 1
0 0 1
1 2 0
2 1 2

1 1 1 0 1 0 1 0
0 1 0 1 2 2 1 0
A language that is not regular

FSAs can’t “count”

Theorem: No finite state machine can decide whether or not its input has the same number of 0’s and 1’s.

Proof:

Suppose an N-state machine does it.
Give it N+1 0’s followed by N+1 1’s
Some state must be revisited
Machine would accept the same string without the intervening 0’s
That string doesn’t have the same number of 0’s and 1’s
Contradiction.

Ex:

```
  .  00000000000000000001111111111111111
  .   x   ...   x
  .
  .  00000011111111111111111
  .   x
```

Need to consider more powerful machines
Add "memory" to the FSA: a pushdown stack
(amount of memory is potentially infinite)

Simple machine with N states (0 to N-1)
start in state 0
read a bit, check bit at top of stack
depending on current state/input bit/stack bit
move to new state
push or pop a bit
stop when last bit read
ACCEPT if stack empty, REJECT otherwise

PDA for deciding whether input has the same
number of 0's and 1's:

PDA's can count, but not much else:
can't recognize set of all strings of the
form AB where both A and B have the
same number of 0's and 1's
more powerful machines needed
Add ability to guess the right answer

Nondeterministic FSA:
Simple machine with N states (0 to N-1)
    start in state 0
    read a bit
    depending on current state, input bit
      move to any of several new states
    stop when last bit read
    ACCEPT if in last state, REJECT otherwise

ACCEPT if there's *some* path to the last state
REJECT if there's *no* path to the last state

Ex:
01
0111110101
01000010110
010000
010000
Nondeterminism doesn’t help in FSAs

Given any nondeterministic FSA, can construct a deterministic FSA that recognizes the same language.

One state in the FSA for every set of states in the NFSA.

Ex.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2 0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>15</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>13</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>15</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>14</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>15</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>4</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0</td>
<td>1</td>
<td>13</td>
<td>6</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>0</td>
<td>14</td>
<td>15</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>15</td>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Any language recognizable by a NFSA is regular.

NFSA $\rightarrow$ FSA $\rightarrow$ RE
FSAs are equivalent to REs

Finite State Automata and Regular Expressions are EQUALLY POWERFUL, or EQUIVALENT

**THM:** For any regular language, there is an FSA that
* accepts strings in the language
* rejects strings not in the language

**proof sketch:**
construct a NFSA, using the following rules:

- a
- AB
- A+B
- B*

**THM:** For any FSA, there is a regular language that
* contains all strings accepted by the FSA
* contains no other strings

[proof omitted]

Equivalence of languages and machine models is essential in the theory of computation
Ex: nondeterministic PDA to recognize set of all strings of the form AB where both A and B have the same number of 0's and 1's.

Construction as for FSAs doesn't work because stack is potentially infinite in size.
Turing machines

Input on a tape that can be read *and* written

Simple machine with N states (0 to N-1)
  start in state 0
  read a bit
  depending on current state, input bit
    write a bit, move tape R or L
    move to a new state
  stop when last bit read
  ACCEPT if in last state, REJECT otherwise

Linear-Bounded Automata:
  length of tape = K * (size of input)

Turing Machine: no restriction on tape

Hierarchy of machines
  Finite-State Automata (-)
  Pushdown Automata (+)
  Linear Bounded Automata (?)
  Turing Machines (-)
(+): nondeterminism helps
(-): nondeterminism does not help
(?): not known whether nondeterminism helps

Each machine is more powerful than the previous
[can recognize more languages]
Are there limits to machine "power"?
Language:
    a set of strings over a finite alphabet

Regular expressions
Finite-state automata
memory
nondeterminism

Given a RE, can construct a FSA that recognizes
    any string matching the RE
Given a FSA, can construct a RE that matches
    the strings recognized by the FSA
Thus, FSA's and RE's computationally equivalent

More powerful machines than FSAs:
    PDA: add pushdown stack, nondeterminism
    LBA: use (linear-bounded) tape
    Turing machine: use infinite tape

What languages do these machines recognize?
Grammars

Generate words in a formal language by a process of replacing symbols systematically

Four elements:
- nonterminals
- terminals
- start symbol
- productions

NONTERMINAL symbols
- "local variables" for internal use
- notation: \(<\text{name}\>\)

TERMINAL symbols
- set of characters that appear in the words "alphabet" of the language
- ex: \{0,1\} or ASCII

START symbol
- one particular nonterminal

Productions
- replacement rules
- ordered pairs of strings of symbols
- notation (ex.): \(a<\text{B}\>c \rightarrow <D>e<\text{F}\>)
- LHS must have at least one nonterminal
A familiar example (abbreviated)

Nonterminal:
   <sentence>  <subject>  <verb>  <object>

Terminal:
   horse  dog  cat  saw  heard  the

Start:
   <sentence>

Productions:
   <sentence>  ->  <subject><verb><object>
   <subject>   ->  the horse
   <subject>   ->  the dog
   <subject>   ->  the cat
   <object>    ->  the horse
   <object>    ->  the dog
   <object>    ->  the cat
   <verb>      ->  saw
   <verb>      ->  heard

"Words" in the language:
   the horse saw the dog
   the dog heard the cat
   the cat saw the horse
   ...

Could write grammar for full English language
   (in principle)
Another familiar example

Nonterminal:

<stmnt> <selection-stmnt> <expression>, etc.

Terminal:

if while else switch ( ), etc.

Start:

<translation-unit>

Productions:

<stmnt>

-> <selection-stmnt>

<selection-stmnt>

-> if ( <expression> ) <stmnt>
-> if ( <expression> ) <stmnt> else <stmnt>

<iteration-stmnt>

-> while ( <expression> ) <stmnt>
-> do <stmnt> while ( <expression> ) ;

A "Word" in the language:

#include <stdio.h>
main()
    { printf("hello, world\n"); } 

Can (in fact) write grammar for full C language
(pp. 234-9 of "The C Programming Language")
Sample grammar

Nonterminal, Start: \(<\text{pal}>\)
Terminals: 0, 1

\(<\text{pal}>\) \rightarrow 0<\text{pal}>0 \mid 1<\text{pal}>1
\(<\text{pal}>\) \rightarrow 0 \mid 1
\(<\text{pal}>\) \rightarrow

DERIVATION: apply replacement rules until no nonterminals left
\(<\text{pal}>\) \rightarrow 0<\text{pal}>0 \rightarrow 01<\text{pal}>10
\quad \rightarrow 011<\text{pal}>110 \rightarrow 0110110

PARSE TREE exhibits derivation

\[
\begin{array}{c}
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. \\
. 
\end{array}
\]

Grammar GENERATES language
set of all strings that could be derived

0, 1, 00, 11, 010, 101, 000, 111, 0000, 00100 . . .
Type 3 grammars

Restrict all productions to be of the form

\[ <X> \rightarrow x \]
\[ <X> \rightarrow <Y>x \]

Ex:

\[ <A> \rightarrow <z>0 \]
\[ <z> \rightarrow <A>1 \]
\[ <z> \rightarrow \]

generates alternating sequence of 0's and 1's

THM: Type 3 grammars are equivalent to REs

proof sketch:

given a Type 3 grammar, construct an FSA that recognizes any string in the language generated by the grammar

given an FSA, construct a Type 3 grammar that generates the strings recognized by the FSA

[FSA states correspond to nonterminals]
Type 2 grammars

Restrict all productions to allow only a single nonterminal on the LHS

Ex:
<pal> → 0<pal>0
<pal> → 1<pal>1
<pal> → 0
<pal> → 1
<pal> →

Ex:

grammar for C

Also called CONTEXT-FREE grammars

Much more descriptive than regular expressions

Q. How does the FSA have to be augmented to recognize strings from languages generated by context-free grammars?

A. Add a "memory" capability (as expected) *and* add power of nondeterminism (!)

Still limited: can't generate strings with equal numbers of a's, b's, and c's
Chomsky hierarchy

Type 1 (context-sensitive) grammars:
add productions of the type
\(<X><Y><Z> \rightarrow <X>y<Z>\)
Type 0 grammars: no restrictions

Essential correspondence between languages and automata

Regular (Type 3)
finite-state machine
Context-free (Type 2)
nondeterministic pushdown automata
Context-sensitive (Type 1)
linear bounded automata
Type 0
Turing machines

One-to-one correspondence to machines persists through the hierarchy
Each type is more descriptive than the previous
Each machine is more powerful than the previous

Are there limits to machine “power”? Are there languages that no machine can recognize? [stay tuned]
Lindenmayer systems

Apply productions simultaneously

Ex:
0 -> 1 [ 0 ] 1 [ 0 ] 0
1 -> 1 1
Start with 0
At stage i, apply rules to each symbol in string form stage i-1

0
1 [ 0 ] 1 [ 0 ] 0
11[1[0]1[0]0]11[1[0]1[0]0]1[0]1[0]0

Applications
model cell growth in biological systems
computer graphics

Not all formal languages fit in Chomsky hierarchy
Graftals

Convert to Lindenmayer system to 2D

1: "stem"
0: "leaf"
[] branch off
   (alternate LR turns along trunk)

\[
\begin{align*}
1 & \quad 1[*]1[*]* \\
11[*]11[*]* & \quad 111[*]111[*]*
\end{align*}
\]

"Graftal" plants
As we make machines more powerful, we can recognize more languages.

Are there languages that no machine can recognize?
Are there limits on the power of machines that we can imagine?

Abstract models of computation help us learn
* nature of machines needed to solve problems
* relationship between languages and machines
* intrinsic difficulty of problems

Automata
Languages
Computability
Complexity

Pioneering work in the 1930's
A Puzzle ("Post's Correspondence Problem")

Given a set of cards
N types of cards, as many as needed
each has a top string and a bottom string

Example 1:

<table>
<thead>
<tr>
<th>BAB</th>
<th>A</th>
<th>AB</th>
<th>BA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ABA</td>
<td>B</td>
<td>B</td>
</tr>
</tbody>
</table>

Puzzle: find a way to arrange the cards so that top and bottom strings are the same (or report that it's impossible).

Solution to Example 1:

<table>
<thead>
<tr>
<th>A</th>
<th>BA</th>
<th>BAB</th>
<th>AB</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABA</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>ABA</td>
</tr>
</tbody>
</table>

Example 2: (no solution)

<table>
<thead>
<tr>
<th>A</th>
<th>ABA</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAB</td>
<td>B</td>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

Surprising fact: this puzzle is UNSOLVABLE! can't write a program to determine whether a given set of cards can be so arranged

How do we know?
Simple machine with $N$ states
start in state 0
read a bit
depending on current state, input bit
write a bit, move tape R or L
move to a new state
stop when last bit read
ACCEPT if in state $X$, REJECT otherwise

Various equivalent definitions
* different conventions for ACCEPT/REJECT
* chars, not bits on tape
* one-way tape

Ex: [Ref. Harel, Algorithmics]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mark</td>
<td>0 3 0 L 1 0 R 2 0 R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>move1</td>
<td>1 5 0 L 1 1 R 1 2 R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>move2</td>
<td>2 6 0 L 2 1 R 2 2 R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>YES</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NO</td>
<td>4 4 0 L 4 1 L 4 2 L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test1</td>
<td>5 3 0 L 7 0 L 4 2 L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test2</td>
<td>6 3 0 L 4 1 L 7 0 L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>back</td>
<td>7 0 0 R 7 1 L 7 2 L</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sample Turing Machine

Stops iff input is a palindrome

Tape:

1. . . 0 0 0 1*2 2 1 0 0 . . mark
2. . . 0 0 0 2*2 1 0 0 . . move1
3. . . 0 0 0 2 2*1 0 0 . . move1
4. . . 0 0 0 2 2 1*0 0 . . move1
5. . . 0 0 0 2 2 1 0*0 . . move1
6. . . 0 0 0 2 2 1*0 0 . . test1
7. . . 0 0 0 2 2*0 0 0 . . back
8. . . 0 0 0 2*2 0 0 0 . . back
9. . . 0 0 0 0*2 2 0 0 0 . . back
10. . . 0 0 0 2*2 0 0 0 . . mark
11. . . 0 0 0 0 2*0 0 0 . . move2
12. . . 0 0 0 0 2 0*0 0 . . move2
13. . . 0 0 0 0 2*0 0 0 . . test2
14. . . 0 0 0 0*0 0 0 0 . . back
15. . . 0 0 0 0 0*0 0 0 . . mark
16. . . 0 0 0 0*0 0 0 0 . . YES
C Program to Simulate Turing Machines

Three-character alphabet (0 is "blank");
Input: description of machine (9 ints/state)
    next state, output char, tape move if 0 read
    next state, output char, tape move if 1 read
    next state, output char, tape move if 2 read

int next[3][100], out[3][100], move[3][100];
char tape[2000]; int head = 1000;
... /* read in machine */
while ((tape[head++] = getchar()) != EOF) ;
state = 0; head = 1000;
while (state != N-1)
{
    in = tape[head] -'0';
    state = next[in][state];
    tape[head] = out[in][state];
    head += move[in][state];
}
printf("Halt ");
}

Detail(?) missing: might run off end of tape
**Machine equivalence**

Turing machines are more powerful than FSA, PDA, LBA because of “infinite” tape memory control triggered from memory.

**CLAIM:** Turing machines are as powerful as real machines

**proof sketch:**
- encode state of memory, PC, etc. on TM tape
- develop TM states for each instruction
can do because all instructions
  - examine current state
  - make well-defined changes depending on current state
could simulate at gate level, machine level,...

**CLAIM:** Turing machines are equivalent to C programs

**proof sketch:**
- C program $\rightarrow$ TOY program $\rightarrow$ TM
- TM $\rightarrow$ C program

Works for all programs and machines?
Universal Turing Machine

A specific Turing machine (UTM) that can simulate the operation of any Turing Machine encode machine on tape

\[ 0 \ 3 \ 0 \ \text{L} \ 1 \ 0 \ \text{R} \ 2 \ 0 \ \text{R} \ 1 \ 5 \ 0 \ \text{L} \ 1 \ 1 \ \text{R} \ 1 \ 2 \ \text{R} \ldots \]

build Turing machine corresponding to C simulator

anything that can be computed by any TM can be computed by UTM

Profound implications

Example:

given new supercomputer X
write "universal" simulator UX
build TM for UX
UTM can compute anything X can
Q. Which problems can a Turing machine solve?
A. *Any* problem *any* computer can solve!

A "thesis", not a "theorem"
can't be proved because we can't precisely define "solving" a "problem" (computability)

Evidence in favor
imagine Turing machines with more "power"
more tapes
2D tapes
nondeterminism(!)
still can simulate with a TM

More evidence in favor
different ways to define "computable"
universal TM
lambda calculus
Post production system
"recursive" functions
all have been proven equivalent

If a problem can't be solved by a TM, we ASSUME that it can't be solved by any other computer.

If a problem can't be solved by *any* specific particular machine, we ASSUME that it can't be solved by any other computer.
Write a C program that reads in another program and its input and decides whether or not it goes into an infinite loop.

**Program 1:**

```c
while (x != 1) {
    if (x > 2) x = x - 2; else x = x + 2;
}
```

**Ex:**

8  6  4  2  4  2  4  2  4  2  4  2  4  2  4  2  4  2  4  2  4  2  4  2  9  7  5  3  1

**Halts iff x odd**

**Program 2:**

```c
while (x != 1) {
    if (x % 2) x = 3*x+1; else x = x/2;
}
```

**Ex:**

8  4  2  1

7  22  11  34  17  52  26  13  40  20  10  5  16  8  4  2  1

???
THEOREM: The halting problem is unsolvable.

Proof:
Consider only programs that print "YES" or "NO".
Assume there exists a program DECIDE that takes any program P as input and prints "YES" if P prints "YES" or prints "NO" if P does not print "YES".
That is, if prints "NO" or doesn't halt.
Change "YES" to "NO" and "NO" to "YES" everywhere in DECIDE to get a program CONFUSE.
The program CONFUSE takes any program P as input and prints "NO" if P prints "YES" or prints "YES" if P prints "NO" or doesn't halt.
Give CONFUSE *itself* as input.
If CONFUSE prints "NO" that means <CONFUSE prints "YES">
If CONFUSE prints "YES" that means <CONFUSE doesn't print "YES">

Contradiction: DECIDE can't exist.

Sketch of proof for halting problem: change "print YES" to "print anything and halt" and "print NO" to "never print anything"
Unsolvable Problems

Halting Problem not "artificial"
reduced to simplest terms to simplify proof
closely related to some practical problems

Very profound implications

Unsolvability of Halting Problem can be
used to show other problems to be unsolvable

Technique: "code" Turing machine into problem
given a TM
create an instance of the problem
with the property that
if there is a solution to the problem
the corresponding TM halts

Examples

Post correspondence problem
Do two programs produce the same input?
Thue's word problem
Equivalence of context-free grammars
Implications

Practical
work with limitations
recognize and avoid unsolvable problems
learn from structure
same theory tells us about efficiency

Philosophical
(caveat: ask a philosopher)
we "assume" that step-by-step reasoning
will solve *any* scientific or technical problem
"not quite" says the halting problem
anything that "is like" (could be) a computer
has the same flaw
physical machine (rods/gears, etc.)
human brain?
matter itself?
universe?
C program
#include <stdio.h>
main()
{
    printf("hello world ");
}

Compile-execute abstraction
cc hello.c
a.out

Systems programs that support this abstraction
    assembler
    macro processor
    compiler (interpreter)
    loader
    linker
    "kernel"
    libraries

Programs that manipulate programs

General approach used
    does not vary much across systems
    developed as consensus of many people
    embodies significant intellectual achievements
    though it makes sense in hindsight
    has had major impact
Assembly Language

Simplify machine language by adding mnemonic codes for opcodes using symbolic names for addresses

Ex:

10: B000 LA R0, 00
11: B11C LA R1, DATA
12: B208 LA R2, SIZE
13: B301 LA R3, 01
14: 2113 S R1, R1 - R3
15: B300 LA R3, 00
16: 9D12 LOOP LI R5, 12
17: 2635 S R6, R3, R5
18: 661A BP R6, * + 2
19: 1305 A R3, R0 + R5
1A: 7216 BC R2, LOOP
1B: 0000 H

SIZE EQ 8
DATA DS SIZE

Advantages over machine language easier to add, remove instructions working with names, not numbers

ASSEMBLER:
program to translate from assembly language to machine language not difficult to implement symbol table (BST?) most difficult part
Low-level languages

Why use assembly language?
   Plus:  good handle on efficiency
   Minus: programs too hard to write
No choice in ancient systems
   resource restrictions
   primitive compilers
   complex abstract machines
Assembly language programming is now rare
With modern compiler technology
   optimized output may be *better* than
   hand-coded version

Issue: is C today’s assembly language?

Disadvantageous similarities
   (language designer’s point of view)
   low level of abstraction
   ability to affect bare machine
   separation from high-level functionality

Advantageous similarities
   (systems programmer’s point of view)
   low level of abstraction
   ability to affect bare machine
   separation from high-level functionality

Note that basic C constructs appear in
   virtually all languages *except* assembly...
Preprocessor (macro processor)

Ref: Kernighan and Ritchie, p. 228

"Prepare" a program for the C compiler

Certain processing functions
not conveniently included in C
still important and useful

Macro definition
#define TABSIZE 100
#define less(A, B) A < B
int table[TABSIZE]
if (less(s, t)) ...

File inclusion
#include <stdio.h>
#include "myfile.c"

Conditional compilation
#define DEBUG 1
#if DEBUG
printf(...);
#endif

INPUT: C program with directives
OUTPUT: C program
low-level, efficient, special purpose program
  can use awk or perl for similar tasks
Compilers

Translate programs
from high-level language
to "native" machine language

Three basic phases
lexical analysis
syntax analysis
code generation

User program:
specification of what user wants

Object module:
realization on a particular machine

application of theory to a practical problem
interaction between languages and machines
nontrivial software engineering
Lexical Analysis

Convert input into stylized stream of "tokens" for later phases

Tokens are TERMINAL SYMBOLS
"letters" in the "alphabet" of the language

Ex: (C)

    if else while do for int float sizeof ...
    { } ; . -> + - * ++ -- ...
    i, j, 123, 12.2, "Hello, world", ...

Lexical analyzers can be implemented with finite-state-machines

Ex:

    sequence of blanks/tabs/etc. -> one blank
    sequence of letters -> "identifier"
    sequence of digits -> "number"

lex: Unix utility to do lexical analysis
    (runs user-defined FSA)
Describe language with a CONTEXT-FREE GRAMMAR
(for C, Kernighan and Ritchie, pp. 234--239)

Ex (prefix expressions):

\[
\begin{align*}
<\text{expr}> & \rightarrow * <\text{expr}> <\text{expr}> \mid <\text{val}> \\
<\text{expr}> & \rightarrow + <\text{expr}> <\text{expr}> \mid <\text{val}> \\
<\text{val}> & \rightarrow \text{identifier} \mid \text{constant}
\end{align*}
\]

RECURSIVE DESCENT parser
subroutine for each nonterminal
subroutine calls for each production
input "match" for each terminal

```c
void expr()
{
    if ( token() == '*' )
    {
        tokenptr++; term(); term();
    }
    ...
}
```

provides structural basis for compilation
extends to create ABSTRACT SYNTAX TREE
Ex: Sedgewick, Program 5.20 (see next slide)

Parsers can be described by and implemented with
pushdown automata
yacc: Unix utility to do syntax analysis
ABSTRACT SYNTAX TREE (parse tree) gives structure of computation

```
  .  *  f
   . +
    . a  *
     . * +
      . b c d e
```

Traverse abstract syntax tree in POSTORDER
      terminal: emit LOAD instruction
      nonterminal: emit ADD or MUL instruction

```
a 9140
b 9241
c 9342
* 3423
d 9243
e 9344
+ 1523
* 3645
+ 1716
f 9145
* 3271
```

Many details omitted (ex: register allocation)
Use intermediate abstract machine
OPTIMIZE code
Compiler-related software tools

INTERPRETER:
High-level language simulation
use *any* convenient abstract machine
output is program output, not translation
Ex:
  TOY machine simulator (lecture 6)
  PostScript
  shell
  java
  emulator (machine-language interpreter)
  usually "slow"
  source instruction \(\rightarrow\)
  many interpreter instructions
Ex: lookup names each time they're used?

COMPILER-COMPILER
program that produces a compiler as output
ultimate "table-driven" compiler
  lexical analysis: regular expression
  syntax analysis: context-free grammar
  code generation: machine description
automatically convert tables into code
useful for
  research on new languages
  bootstrapping (first compiler)
What is "a.out", exactly?

COMPILER:  C \rightarrow assembly language
ASSEMBLER:  assembly \rightarrow relocatable object code
LOADER:  relocatable object code \rightarrow executable

Addresses in code depend upon where it is loaded

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B000</td>
<td>LA R0, 00</td>
<td></td>
</tr>
<tr>
<td>B11C</td>
<td>LA R1, DATA</td>
<td></td>
</tr>
<tr>
<td>B208</td>
<td>LA R2, SIZE</td>
<td></td>
</tr>
<tr>
<td>B301</td>
<td>LA R3, 01</td>
<td></td>
</tr>
<tr>
<td>2113</td>
<td>S R1, R1 - R3</td>
<td></td>
</tr>
<tr>
<td>B300</td>
<td>LA R3, 00</td>
<td></td>
</tr>
<tr>
<td>9D12</td>
<td>LOOP</td>
<td></td>
</tr>
<tr>
<td>2635</td>
<td>S R6, R3, R5</td>
<td></td>
</tr>
<tr>
<td>661A</td>
<td>BP R6, * + 2</td>
<td></td>
</tr>
<tr>
<td>1305</td>
<td>A R3, R0 + R5</td>
<td></td>
</tr>
<tr>
<td>7216</td>
<td>BC R2, LOOP</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>H</td>
<td></td>
</tr>
</tbody>
</table>

Compiler can't know these values

RELOCATABLE OBJECT CODE
machine language code PLUS tables giving which addresses are RELATIVE to start
LOADER translates object code to executable code by adding start to each relative address.
Modern operating systems support
* software tools for creating programs (lect. 18)
* running multiple programs (multiprogramming)
* saving/accessing data (files, virtual memory)
* user interaction (window system)
* interaction with other systems (networking)
* core applications programs

Software libraries
Multiprogramming
File system/virtual memory
Window manager
  * single-user multiprogramming
  * graphic user interface
Applications
Networking

**EXECUTION CONTROL**
  OS keeps track of state of CPU, devices

**EXTERNAL DEVICES**
  Display, keyboard, mouse, disks, CD, network

**VIRTUAL MACHINES**
  OS implements abstract devices
    * simpler for user than real hardware
    * easier to change to new hardware
Libraries and separate compilation

Programs use LIBRARIES or other modules ex: printf
Recompile library program when it is used?

Compiler produces OBJECT MODULE
* code
* addresses for variables defined within
  * external variable references

RELOCATE: add start address to each instruction address field
RESOLVE: find actual addresses for each external variable, fill in at each place marked by the compiler

LINKER (link editor)
* concatenates input object modules to make a single output file
* resolves external references
* merges relocation tables

\[ \text{a.out} = \quad \text{code} \quad + \text{relocation tables} \]

LOADER translates object code to executable code by adding start address to each reference requiring relocation (info in merged table)
Operating system "kernel"
    keeps track of several programs

**INTERRUPT:**

    stop
    save PC somewhere "special"
    change PC

Interrupts are
    necessary to manage input-output devices
    useful for multiprogramming

OS allows several programs to "share" CPU by
    keeping table of "current" PC's for programs
    setting clock to interrupt periodically

**RELOCATABLE** program:
    can be moved while it is executing
    (useful if OS rearranges memory a la malloc)

**REENTRANT** program:
    can be executed while it is executing
    (only load one copy of program; e.g. emacs)
UNIX file system layout

Goal: provide simple abstraction
(sequence of bytes) for user programs

Each disk has
"index" information (i-nodes)
"data" (data blocks)

Superblock (block 1)
size and number of data blocks
size and number of i-nodes
free list of data blocks

I-nodes (one per file)
accounting info
pointers to disk blocks
(tree structure for big files)

Data blocks: just data
Directory: list of file names/i-node addrs
File: list of data blocks

forms a TREE
traverse the tree for sequential access
File layout examples

Small file:
  i-node lists data blocks
  Ex: 10 i-node entries, 1K data blocks
  handles files < 10K

Medium-sized file:
  i-node lists blocks that list data blocks
  Ex: 10 i-node entries,
      256 data block pointers/block
  handles files < 2.56 M

Large files:
  add a third level
  Ex: 10x256x256x1K = 655.36 M

Typical medium-sized file

Tradeoff on data block size
  Too small: Large files excessively fragmented
  Too large: Excess waste in small files
Virtual Memory

Problem 1:
several programs need to share same memory
direct solution: divide up the memory

Problem 2:
program needs more memory than machine has
direct solution: “overlays”

Problem 3:
it’s all just memory
why should file system look more complicated?

“Better” solution to all three problems:
all programs assume access to all memory
each program actually uses a small portion

VIRTUAL MACHINES
simulate multiple copies of a machine on itself
Ex: can debug OS

physical address space
how much memory is there?
limitation: $/bit cost

virtual address space
maximum amount of memory
an instruction could directly reference
limitation: address size (bits/instruction)
Paging

Widely-used method for implementing VM

Design hardware to "trap" all addresses
Keep virtual memory (for each program) on disk

Divide into PAGES
Keep table with
* flag indicating if page is in memory
* relative position of page in memory

Make page size a power of two,
use leading bits of address for page name
Each memory reference
check if page is in memory
get it from disk if not
use page table to reset upper address bits
Each page brought in has to REPLACE another
"page replacement strategies" still being studied, invented

Basic principles
MEMORY HIERARCHY
local: fast, small, expensive
remote: slow, huge, cheap
CACHE recently accessed information
to give illusion that all info is local
Size of Virtual Memory

16 bits is not enough
24 bits is not enough
32 bits is not enough

Is 64 bits enough?
18446744073709551616 > 10^{19} addresses

Need more sophisticated paging strategy, primarily because page table would be too big
multilevel
associative

Some big numbers
10^{20}: number of grains of sand
10^{27}: number of oxygen atoms in a thimble
2^{256}: number of electrons in the universe

512 bits is certainly enough
no need for relocation,
use a galactic address dispenser
Window Manager

VIRTUAL TERMINALS

X-terminal
  complex
  customizable
  virtual!

Just another simulation program

Commonplace today, rare in 1985
ingenious design meets accelerating technology

History
  Xerox PARC (Alto)
  Macintosh
  Windows NT
  X-terminal
  Netscape

Problem or opportunity?
  truly "virtual"
  moving away from grounding in reality
  flexibility vs. standardization
  other ways of interacting with computer?
Client-Server Model

System divided into two distinct parts

DISPLAY SERVERS
  present display
  monitor keyboard and mouse input
  (implement virtual display)

CLIENTS
  applications programs
  (use virtual display)

Server is interface between
  client program and display hardware

Model generalizes beyond display management
  client: request service
  server: do the work
  Ex: file service

Advantages
  single server can handle multiple clients
  keeps kernel simple, adaptable
  smooth transition to DISTRIBUTED SYSTEM
The Network

Ultimate distributed system

INTERNET: "all the cooperating networks"

Circuit switched network: phone system
Packet switched network: network system

IP: Internet protocol
   packet:
      1-1500 bytes
      from address
to address
   address:
      ex. 128.112.128.43

ROUTERS move packets across network

TCP: Transmission control protocol
   break big messages into packets
   collect received packets into messages
   check for errors

Domain Name System
   distribute authority/responsibility
      for name service
   can use "phoenix.princeton.edu"
   not 128.112.128.43 (/etc/hosts)

(many details omitted!)
Operating systems/network issues

Network applications
  communication (mail, news)
  remote login (telnet)
  file transfer (ftp)
  publishing (html)
  browsing (netscape)
  e-commerce

Modern rendition of ancient tradeoffs
  Personal computer or Network computer
  ONE huge virtual machine?!

compare/contrast
  * computer center
  * phone system
  * post office (snail mail)
  * libraries

Current network ethics
  * honor and foster individualism
  * the network is good and must be preserved

Should hackers or the government “run” the net?
  * can commercial apps trust an “open net”?
  * does a “closed net” violate individual rights

Security/Privacy/Copyright

Who owns??
Who pays??
Operating systems (mass market, today)
set of applications programs
  mail
  editor
  database
  spreadsheet
  desktop publishing
  internet browsing/publishing
  ....

Drawback: applications are for
corporate America
home computing

Examples of basic tools for scientists/engineers
  emacs
  netscape/AltaVista (HTML)
  Maple/Mathematica
  awk/perl
  TeX/LaTeX
  PostScript
  C/C++/Java/Fortran

COS 126 skills give you "license" to use these
converse not true!
[be aware of unlicensed users]
Use the mass-market tools, too
[when they're better!]
Emacs

. Emacs Makes A Computer Slow
. Escape Meta Alt Control Shift
. Emacs Makers Are Crazy Sickos
. Emacs Makes All Computing Simple
. Emacs Makefiles Annihilate C-Shells
. Emacs May Allow Customized Screwups
. Emacs Manuals Are Cryptic and Surreal
. Eventually Munches All Computer Storage
. Eight Megabytes And Constantly Swapping
. Elsewhere Maybe All Commands are Simple
. Excellent Manuals Are Clearly Suppressed
. Emacs May Alienate Clients and Supporters
. Extended Macros Are Considered Superfluous
. Except by Middle Aged Computer Scientists
. Every Mode Accelerates Creation of Software

Features

. full set of basic editing commands
. efficient keystroke codes
. extensible/customizable

Bottom line: efficient person-machine interaction

. try out various capabilities
. buy a manual
. surf the web

www.cgd.ucar.edu/gds/thibaud/Emacs/slides.html
**HTML**

Make yourself a home page:

```bash
mkdir public_html
chmod 0755 public_html
cd public_html
emacs index.html
```

Put the following in index.html

```html
<HTML>
<HEAD><TITLE>Your Name</TITLE></HEAD>
<BODY>
<H2>Your Name</H2>
<p>Anything else you’d like to say</p>
</BODY>
</HTML>
```

```bash
chmod 0755 index.html
```

In Netscape, open

`www.princeton.~username`

Voila!

Next steps:

- add stuff (next slide); debug with "Reload"
- copy and edit someone else’s (it’s public!)
- surf the web (not worth buying manual)
- step-by-step introduction

http://www.CS.Princeton.EDU/courses/archive/fall97/cs111/labs/introHTML

online manual

http://werbach.com/barebones/barebone_html.html
MARKUP:

- metacommands specifying document structure
- fonts
  `<b>bold</b>`  `<i>italic</i>`  `<tt>typewriter</tt>`
- headers
  `<h3>hypertext markup language</h3>`
- paragraph
  `<p>
  It was the best of times; it was the worst of times
  </p>`
- itemized list
  `<ul>
  <li>headers
  <li>paragraph
  <li>itemized list
  </ul>`

HYPERTEXT: text with hot links

`<A HREF="URL"></A>`

- tables, graphics

Defacto standard: 1950s technology

ISSUES

* who writes HTML code? [person or program?]
* who decides appearance? [author or browser?]
* can we have a different standard? [too late?]
* content vs. form
Mathematical Applications

"Let the computer do it!"
numerical computation
symbolic manipulation
graphs/plots

Many systems available
tailored to application or general purpose?
worth using new environment
for problem at hand?
interfaces to systems in use
quality of user interface
extent of mathematical library
compute the right answer?
available on your machine?

C, FORTRAN, Java (with old code and libraries)
bc, dc
maple, mathematica, matlab

Wolfram, "Mathematica, A system
for doing math by computer"
Char, Geddes, Leong, Gonnet, et. al.
"Maple V Reference Manuals"
"First Leaves: Tutorial Intro to Maple V"
numeric
> 2+2;

4

> p := x -> x^3+x^2-3*x +2;

p := x -> x^3 + x^2 - 3*x + 2

> p(2.3333);

13.14754816

symbolic
> p(y^2+3);

(y^2 + 3)^3 + (y^2 + 3)^2 - 3(y^2 + 3) - 7

> simplify(");

6 y^4 + 10 y^3 + 30 y^2 + 29

typical sequence
> int(p(x)*cos(x),x);

. x sin(x) + 3 cos(x) x - 9 cos(x)
. - 9 x sin(x) + x sin(x) + 2 cos(x) x

> subs(x=2,");

-6 sin(2) + 7 cos(2)

> evalf(");

-8.368812417

> quit;
evaluate functions
> evalf(cot(.6 + Pi/8));

.6524523271

find roots
> fsolve(cot(x+Pi/8) = x, x);

.6218013732

number theory
> ifactor(1010101010101010101010);

(2) (5) (11) (23) (513239) (21649) (8779)

> 2*5*11*23*513239*21649*8779*4093;

1010101010101010101010

linear algebra
> A := array([[1,3,-4], [1,1,-2], [-1,-2,5]]);

[ 1  3  -4 ]
[            ]
A := [  1  1  -2 ]
[            ]
[ -1  -2   5 ]

> Ainv := evalm(A^(-1));

[ -1/4   7/4  1/2 ]
[                 ]
Ainv := [  3/4  -1/4  1/2 ]
[                 ]
[  1/4   1/4  1/2 ]

Jacobian, Laplacian, etc.
Symbolic computation (Maple)

root-finding
> solve( x^4 - 7*x^3 + 3*a*x^2 = 0, x );
 0, 0, 7/2+1/2 (49-12 a) , 7/2-1/2 (49-12 a)

differential equations
> diff(y(x),x,x) + 2*diff(y(x),x) + y(x) = exp(-x)
 2
  d      d
  ---- y(x) + 2 -- y(x) + y(x) = exp(- x)
  dx     dx
> dsolve({"", y(0)=1, D(y)(0)=0}, y(x));
 2
  y(x) = 1/2 x  exp(- x) + exp(- x) + exp(- x) x

series manipulations
> s := series(sin(x), x=0);
 3      5       6
  s := x - 1/6 x  + 1/120 x  + O(x )
> c := series(cos(x), x=0);
 2      4       6
  c := 1 - 1/2 x  + 1/24 x  + O(x )
> t := series(s/c, x=0);
 3      5       6
  t := x + 1/3 x  + 2/15 x  + O(x )
Printing and typesetting

Typewriter
Keypunch + card reader + impact line printer
Traditional
  hot lead, galley proofs, etc.
WYSIWYG
  bitmap display editor
  laser printer
  ex: Bravo (1977) MacWrite, Word
"Batch"
  edited typescript
  typesetting software
  phototypesetter
  ex: vi + troff (1970x)
  [emacs, etc.] + TeX (1979)

Internet publishing, hypertext

ALL INVOLVE: keyboard for input

Special challenges
  mathematical typesetting (TeX)
  hypertext (HTML)
  integrated fonts, graphics (PostScript)
  graphic art (Illustrator, Photoshop)
  standard representation (.pdf files)

Paperless, someday?
Ref: The TeXbook, Knuth.

**INPUT:** markup language

- control sequences \xxx
- grouping \{ \}
- macros \def
- math formulae $$
- flexible space \hskip \vskip
- font control \rm \it \bf
- tables, alignments \halign
- page description \eject

"programming" can be incidental
(but general functionality available)

**OUTPUT:**

- long list of \((x, y, \text{character})\) records

**advantages**

- faithful to "art" of typsetting mathematics
- box-and-glue metaphor the "right" one

**criticisms**

- bitmap, nonstandard fonts
- no graphics
- baroque markup language (even LaTeX)
- debugging, interaction weak

No reasonable alternative
for mathematical/technical publishing
Boxes and Glue

"Stuff" in TeX is in boxes, connected by glue

Horizontal, vertical, recursive

Flexible

User specifies "degree" of flexibility

System "sets" the glue to do the typesetting nontrivial algorithms
Polynomial multiplication is easily accomplished with term-by-term multiplication:

$$(x^0+x^2)(x^3+x^5) = x^3 + 2x^5 + x^7$$

but it could be even more easily accomplished if a program were to do it. The program should collect terms, otherwise it might produce, for example, $x^3 + x^5 + x^5 + x^7$ as the expanded version of the product $(x^0 + x^2)(x^3 + x^5)$; and it should eliminate terms with 0 coefficients, otherwise it might create extraneous output for an example like

$$(x^1-x^2)(x^1+x^2) = x^2 - x^4.$$  

There are plenty of other conditions that need to be checked. We could also differentiate:

$$\frac{d}{dx}(x^3 + 2x^5) = 3x^2 + 10x^4;$$

or integrate:

$$\int(x^3 + 2x^5) = \frac{x^4}{4} + \frac{x^6}{3}.$$  

These are some of the situations that a program for symbolically manipulating polynomials might have to handle.
Sample TeX output

Result of
  tex sample.tex
  lpr -d sample.dvi
Perspective on math applications

Use maple, matlab or mathematica, regularly!
- linear algebra
- elementary calculus
- statistics
- power series
- number theory
- linear programming (simplex)
- orthogonal polynomials

Indispensible tools in applied mathematics
- in widespread use
- can't be effective without such software

"Handbook of Math Functions" (and much more)
- now encoded in such software.

System development = math research
- Ex: There exists an algorithm for integration
- Can "write programs" in Maple/Mathematica
- same basic control primitives as C
- hard to notice difference in small programs
- stay within environment
- *but* switch to C if performance-critical

ISSUE: Are "hard problems" solved properly?
- still have to understand the computation

Use TeX to communicate results
- (.tex, .dvi, .ps, or .pdf on web)
Java is
  simple
  object oriented
  statically typed
  architecture neutral
  multi-threaded
  garbage-collecting
  robust
  small
  fast
  secure
  extensible
  well-understood
  fun
-----------------
not new
designed for toasters
not done yet
not as useful as C++, C, FORTRAN, ...
slow
unsafe
huge
complex

Don't believe anything on this slide!
[make up your own mind]
Java perspective

Similar to C in many ways
  int, float, double, char ...
  for, if, while, do, switch ...

Modern primitives
  Ex: Unicode
    16-bit "universal" character set (65536 chars)

Higher level of abstraction than C
  object-oriented
  no pointers
    (references to objects)
  inheritance
  language support for
    interfaces
    multiple threads
    exceptions
  system support for
    software libraries
    security
    graphic user interface
    network

Ref: Arnold, Gosling "Java Programming Language"
    Addison-Wesley, 1996
    http://java.sun.com
"hello, world" in C

```c
#include <stdio.h>
main()
{
    printf("hello, world ");
}
```

COMPILE and EXECUTE the program

```
% cc hello.c
% a.out
hello, world
```

"cc" COMPILES source into machine language
argument: hello.c
output: a.out
"a.out" LOADS and EXECUTES that program
output: [result of running program]

advantage
good compilers make efficient programs

disadvantage
translated code is machine-dependent
class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World");
    }
}

Make sure java is installed. For arizona:
% which javac
no javac in /u/rs/bin /usr/princeton/bin ...
% set path = (/opt/jdk1.1.1/bin $path)

TRANSLATE and INTERPRET the program
% javac HelloWorld.java
% java HelloWorld
Hello, world

"javac" TRANSLATES source into byte stream
argument: HelloWorld.java
output: HelloWorld.class

"java" INTERPRETS byte stream on host machine
argument: HelloWorld.class
output: [result of running program]

advantage: architecture neutral
disadvantage:
    interpretation slower than compiled code
good native java compilers inevitable
#include <stdio.h>
float epsilon = .000001;
float f(float x)
    { return 2.0 - x*x; }
float root(float l, float r)
    {
        float m;
        while (r - l > epsilon)
        {
            m = (l + r)/2;
            if (f(m) > 0) l = m;
            if (f(m) < 0) r = m;
            printf("%f %f ", l, r);
        }
        return (l + r)/2;
    }
main()
    {
        printf("%f ", root(0.0, 2.0));
    }
Square root (bisection method) in Java

Can mimic C code with STATIC functions
"function-oriented" programming

class Bisect
{
    static final double epsilon = .000001;
    static double f(double x)
    {
        return 2.0 - x*x;
    }
    static double root(double l, double r)
    {
        double m;
        while (r - l > epsilon)
        {
            m = (l + r)/2;
            if (f(m) > 0) l = m;
            if (f(m) < 0) r = m;
            System.out.println(l + " " + r);
        }
        return (l + r)/2;
    }
    public static void main(String[] args)
    {
        System.out.println(root(0.0, 2.0));
    }
}

"main" must be public and static
% od -c HelloWorld.class
. 000 312 376 272 276 \0 003 \0 - \0 \b \0 024 007 \0 025
. 020 007 \0 032 007 \0 033 007 \0 034 \n \0 004 \0 \t \t \0
. 040 005 \0 \n \n \0 003 \0 013 \f \0 020 \0 \f \f \0 036
. 060 \0 027 \f \0 037 \0 \r 001 \0 003 ( ) V 001 \0 025
. 100 ( L java / lang / String ; ) V 001 \0 026 ( [ L java /
. 140 lang / String ; ) V 001 \0
. 160 021 0 0 HelloWorld . java
. 200 va 001 \0 006 < init > 001 \0 004 Const
. 220 de 001 \0 \r Constant Val
. 240 ue 001 \0 \n Exceptions 001
. 260 \0 \f HelloWorld , world 001 \0
. 300 \n HelloWorld 001 \0 017 Li
. 320 ne Number Table 001 \0 025
. 340 L java / io / PrintSt
. 360 rem ; 001 \0 016 Local Var
. 400 iables 001 \0 \n Source F
. 420 ile 001 \0 023 java / io / Pr
. 440 int Stream 001 \0 020 java
. 460 / lang / Object 001 \0 020 j
. 500 ava / lang / System 001
. 520 \0 004 main 001 \0 003 out 001 \0 007 p
. 540 rintln \0 \0 002 \0 004 \0 \0 \0 \0
. 560 \0 002 \0 \t \0 035 \0 016 \0 001 \0 021 \0 \0 \0 %
. 600 \0 002 \0 001 \0 \0 \0 \t 262 \0 007 022 001 266 \0 \b
. 620 261 \0 \0 \0 001 \0 026 \0 \0 \0 \n \0 002 \0 \0 \0 \0
. 640 005 \0 \b \0 003 \0 \0 \0 020 \0 \f \0 001 \0 021 \0
. 660 \0 \0 035 \0 001 \0 001 \0 0 \0 \0 005 * 267 \0 006 261
. 700 \0 \0 \0 001 \0 026 \0 \0 \0 006 \0 001 \0 \0 \0 \0 \0
. 720 \0 001 \0 031 \0 \0 \0 \0 002 \0 017

Names, strings, ops in bytecode actually enough info to DECOMPILE code

www.ahpah.com
Object-oriented programming

Direct support of "data type" concept
encapsulate functions that manipulate data
build levels of abstraction

CLASS

data type descriptions
definition of associated functions (METHODS)

OBJECT

"instance" of class
data values describe object "state"
[each object has its own values]
methods describe object "behavior"
[methods are shared with other
objects in the class]

METHOD (function)

instantiates objects
invoke other methods

objects are passed by REFERENCE

strong TYPE CHECKING

types of method args, return must match

OVERLOADING: if arg types differ, methods differ

compare with C "user-defined types"
difference:

uniform model, supported by language/system

Java program: set of classes
Class implementation for rational numbers

Class Rational defines
data type (two integers)
associated methods (mul and add)
CONSTRUCTOR (create a new object)
"toString" method (for output)
[see programming assignment 3]
class Rational
{
    private int p, q;
    Rational(int p, int q)
    { this.p = p; this.q = q; }
    public String toString()
    {
        return Integer.toString(p) + "/" + Integer.toString(q);
    }
    void mul(Rational x)
    { this.p *= x.p; this.q *= x.q; }
    void add(Rational x)
    {
        this.p = this.p*x.q + x.p*this.q;
        this.q *= x.q;
    }
}
"+" with string args: CONCATENATES them
Example Rational client

to declare variables
  of type "reference to Rational"
     Rational t;

to create a new Rational
     Rational e = new Rational(1, 1);

to invoke a method
     e.add(t);

"this" in code refers to e's copy of data values
if p and q were "public", could say e.p, e.q

class e
{
    public static void main(String[] args)
    {
        int i;
        Rational e = new Rational(1, 1);
        Rational t = new Rational(1, 1);
        for (i = 0; i < 6; i++)
        {
            t.mul(new Rational(1, i+1));
            e.add(t);
            System.out.println(e + " " + t);
        }
    }
}
Inheritance

Easy to OVERRIDE or add new methods

class RationalA extends Rational
{
    RationalA(int p, int q)
    { super(p, q); }
    private static int gcd(int m, int n)
    {
        if (n == 0) return m;
        return gcd(n, m % n);
    }
    private static void reduce(Rational x)
    { int t = gcd(x.p, x.q);
        x.p /= t; x.q /= t;
    }
    void add(Rational x)
    {
        p = this.p*x.q + x.p*this.q;
        this.q *= x.q;
        reduce(this);
    }
}

ABSTRACT CLASSES and INTERFACES
no implementations in superclass
subclasses provide implementations
hierarchy can get complicated
support client-interface-implementation paradigm
Why Java?

Basic features present in older languages
- C++
- Modula-3
- Smalltalk
- Ada
- ...

What distinguishes Java?
- free
- easy to install (?)
- works on most machines
- modern, direct full-system approach to software engineering
- growing library of software
- internet applications
  - applets
  - graphical user interface
    [stay tuned]
- it *is* fun

SECURITY
- run programs in Netscape
- can “check” that they won’t crash (?)

Bumps in the road
- huge amounts of old software
- libraries too complex
- disagreement on “standards”
  [check what happened to older languages!]
Object-oriented programming review
Applets
Java and HTML

appletviewer
Netscape

API: Application Programming Interface
AWT: Abstract Windows Toolkit
Sample class: Cartesian points

A java class is a user-defined data type
class Point
{
    public double x, y;
    Point(double xval, double yval)
    { x = xval; y = yval; }
    public static double dist(Point a, Point b)
    {
        double dx = a.x - b.x, dy = a.y - b.y;
        return Math.sqrt(dx*dx + dy*dy);
    }
}

to declare variables
    of type "reference to Point"
    Point a, b, p;
to create a new Point
t = new Point(x, y);
to access fields
double dx = a.x - b.x, dy = a.y - b.y;
to access static method "dist"
delta = Point.dist(a, p);
Alternate design for Point class

Pure object-oriented programming approach
associate "dist" method with point objects

class Point
{
    public double x, y;
    Point(double xval, double yval)
    { x = xval; y = yval; }
    public double dist(Point b)
    {
        double dx = this.x - b.x;
        double dy = this.y - b.y;
        return Math.sqrt(dx*dx + dy*dy);
    }
}

to compute dist from a to p
    delta = a.dist(p);

or
    delta = p.dist(a);
in implementation,
special keyword "this" refers to
object used to invoke method

When should a method be static?
design decision
[some people have religious beliefs]
Linked structures

No explicit pointers (!)  
*but* implicit pointers are everywhere

Ex: linked list of nodes containing Points

```java
class Node {
    public Point p;
    public Node next;
    Node(Point pt)
        { p = pt; next = null; }
}
```
to declare variables  
of type "reference to Node"

Node s, t, mins, tour = null;
to create a node containing  
(reference to) Point p:

t = new Node(p);
to access point in a given node:

a = s.p;
to follow a link:

b = s.next.p;

AUTOMATIC GARBAGE COLLECTION  
since you can't manipulate pointers  
system "knows" when there are no active  
references to an object
Object-oriented programming makes it natural to express computation by building layers of abstraction.

Ex: (from code for TSP assignment)

linked list of Nodes with Points
function to compute total dist
static double tourdist(Node t)
{
    Node s;
    double sum = Point.dist(t.p, t.next.p);
    for (s = t.next; s != t; s = s.next)
        sum += Point.dist(s.p, s.next.p);
    return sum;
}

function to compute distance increase
static double delta(Node s, Point p)
{
    Point a = s.p, b = s.next.p;
    return Point.dist(a, p) + Point.dist(p, b)
    - Point.dist(a, b);
}

exercise: make Tour itself a class

Writing such code is *possible* in C
*supported* in Java (and C++)
Applications Programmers Interface
huge library of classes
Ref: Gosling, Yellin, Java Team
java.sun.com/products/
  jdk/1.1/docs/api/packages.html

PACKAGE: set of classes
API packages
java.lang
  container classes: Integer, Double, ...
  language extensions: String, Class, ...
  Java Virtual Machine: threads, ...
  system resources: Math, in, out, ...
java.io
  basic I/O support
java.util
  generic data structures
    Random, Time, Date...
java.net
  URLs, IP addresses, Sockets, ...
Abstract Windows Toolkit (AWT)
java.awt
  Graphics
java.applet
  Applet

buyer beware:
documentation is produced automatically
Using the API

Scan documentation/manuals for what you need
find a package
find a class
object-oriented or static method
Import to resolve names if necessary

Ex: Two ways to compute random numbers

FUNCTION-ORIENTED (static method)

    double x = Math.random();

OBJECT-ORIENTED:

    control over state
    multiple generators in one program
    import java.util.Random;
    ...
    Random rand = new Random();

rand: OBJECT that can generate random numbers
instance of Random class
contains state information, methods
    double x = rand.nextDouble();

Reasons to use API

    basic language extensions
        (Math, String, IO, ...)
    network and graphics abstractions
        (abstract window toolkit, applets)
Traveling Salesperson solution (assignment 4)
user-defined resources: Node, Point
system resources: parse, random, print
class TSP
{
  // delta and showtour
public static void main(String[] args)
{
  double min, del;
  int N = Integer.parseInt(args[0]);
  Point p, Node s, t, mins, tour = null;
  for (int i = 0; i < N; i++)
  {
    p = new Point(Math.random(),Math.random());
    t = new Node(p);
    if (tour == null) { tour = t; tour.next = t;
      mins = tour; min = delta(mins, p);
      for (s = tour.next; s != tour; s = s.next)
        if ((del = delta(s, p)) < min)
          { mins = s; min = del; }
      t.next = mins.next; mins.next = t;
    }
    System.out.println("Length "+tourdist(tour));
  }
}
is this "fun", yet?
Everyone's first applet

HelloWorldApp.java:
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorldApp extends Applet {
    public void paint(Graphics page) {
        page.drawString("Hello world!", 50, 25);
        page.fillRect(50, 30, 85, 4);
    }
}
NOTE: "paint" not "main" (main is in Applet)
"javac HelloWorldApp.java"
makes HelloWorldApp.class, as usual

HelloWorld.html:
<title>Hello, World</title>
<hr>
<applet
code=HelloWorldApp.class
width=577 height=400>
</applet>
<hr>
Open "HelloWorld.html" in netscape: voila!
Use "appletviewer HelloWorld.html" to debug
Recursive graphics (ref. prog. assignment 5)

```java
import java.applet.Applet;
import java.awt.Graphics;
public class Star extends Applet
{
    static final int N = 128;
    static int SZ = 512/(N+N);
    static Graphics g;
    public static void star(int x, int y, int r)
    { int sz = 2*SZ*r-1;
        if (r ,= 0) return;
        g.fillRect(SZ*(x-r), SZ*(y-r), sz, sz);
        star(x-r, y+r, r/2);
        star(x+r, y+r, r/2);
        star(x-r, y-r, r/2);
        star(x+r, y-r, r/2);
    }
    public void paint(Graphics window)
    {
        g = window;
        star(N, N, N/2);
    }
}
```
TSP applet

Add graphics to TSP program in four easy steps

* replace

```java
class TSP

    import java.applet.Applet;
    import java.awt.Graphics;
    public class TSPapplet extends Applet

* add new method to draw path segments

    static void drawpath
        (Point p, Point q, Graphics page, Color c)
    {
        int xp = (int)(512*p.x), yp = (int)(512*p.y);
        int xq = (int)(512*q.x), yq = (int)(512*q.y);
        page.setColor(c);
        page.drawLine(xp, yp, xq, yq);
    }

* replace

    public static void main(String[] args)

with

    public void paint(Graphics page)

* add code in loop to update path

    drawpath(mins.p, t.next.p, page, Color.white);
    drawpath(mins.p, t.p, page, Color.black);
    drawpath(t.p, t.next.p, page, Color.black);

Exercise: try this out!

    javac TSP.java
    appletviewer TSP
Java summary

C-like language for basic programs

object-oriented programming
  direct data type support
  build layers of abstraction

libraries
  language extensions
  graphics user interface
  networks

advanced concepts
  abstract classes
  interfaces
  Java Virtual Machine
    threads
    exceptions
    sockets
    ...

visit
  http://java.sun.com
regularly to appreciate scope of effort

serious alternative: C++
Ref: Stroustrup, C++ Programming Language
ALGORITHM

“method” used to solve a problem

generally independent of

machine, programming language

DESIGN: find a way to solve the problem

ANALYSIS: determine its effectiveness

New machine

costs $$$ or more

makes “everything” finish sooner

may not help much with some specific problem

(but could)

New algorithm

costs $ or less

may make the difference

allowing specific problem to be solved

may not help much with some other problems

(but could)

Ex: N-body simulation

brute-force N^2 method

N lg N algorithm enables new research
PROBLEM: Given an array of integers, rearrange them so that they are in increasing order.

Ex: sort program with driver, in C

```c
#include <stdio.h>
typedef int Item;
#define less(A, B) (A < B)
#define exch(A, B) { Item t = A; A = B; B = t; }
#define compexch(A, B) if (less(B, A)) exch(A, B)
void sort(Item a[], int l, int r)
{
    int i, j;
    for (i = l+1; i <= r; i++)
        for (j = i; j > l; j--)
            compexch(a[j-1], a[j]);
}
main(int argc, char *argv[])
{
    int i, N = atoi(argv[1]), pr = atoi(argv[2]);
    int *a = malloc(N*sizeof(int));
    for (i = 0; i < N; i++)
        a[i] = 10000*(1.0*rand()/RAND_MAX);
    sort(a, 0, N-1);
    if (pr)
        for (i = 0; i < N; i++)
            printf("%4d ", a[i]);
}
[see Program 6.1]
```
Insertion sort

Brute-force sorting solution
* move left-to-right through array
* exchange next element with larger elements to its left, one-by-one

```
void insertion(Item a[], int l, int r)
{
    int i, j;
    for (i = l+1; i <= r; i++)
        for (j = i; j > l; j--)
            compexch(a[j-1], a[j]);
}
```
Proiling insertion sort

Use lcc "profiling" capability to get frequency-of-execution counts

lcc -b insertion.c
a.out 1000 0
bprint

makes a file "prof.out" that has
the counts for each instruction

void insertionsort(Item a[], int l, int r)
  { int i, j;
    for (i = l+1; i <= r; i++)
      for (j = i; j > l; j--)
        compexch(a[j-1], a[j]);
  }

Striking feature: HUGE numbers

To sort N items,
insertion sort does about \(\frac{N^2}{2}\) comparisons

running time is QUADRATIC
[proportional to \(N^2\)]
Estimating the run time

Total run time =
    sum over all instructions: frequency * cost
frequency:
* determined by algorithm and input
* lcc -b (or analysis) can help estimate
cost:
* determined by compiler and machine
* could estimate by lcc -S (plus manuals)

Easier alternative
1. analyze asymptotic growth
2. for small N, run and measure time
3. for large N, use 1 and 2 to predict time

asymptotic growth rate:
    estimate time as function of input size
ex: N, N*log N, N^2, N^3, 2^N

Ex:
insertion sort time is O(N^2)
takes about .1 sec for N = 1000
how long for N = 10000 ?
    about 100 times as long (10 sec)
how long for N = 1 million ?
    another factor of 10^4 (1.1 days)
how long for N = 1 billion ?
    another factor of 10^6 (31 centuries)
Quicksort

Divide-and-conquer sorting method

To sort an array, first divide it so that
* some element $a[i]$ is in its final position
* no larger element left of $i$
* no smaller element right of $i$

Then sort the left and right parts recursively
Partitioning

To partition an array, pick a partitioning element
* scan from right for smaller element
* scan from left for larger element
* exchange
* repeat until pointers cross

ASORTINGEXAMPLER

ASAMPLE
AASAMPLE
OEXOSAMPLE
ARERTING
AAEETINGEROXSAMPLE

23-7
Partitioning implementation

\[ v: \text{partitioning element} \]
\[ i: \text{left-to-right pointer} \]
\[ j: \text{right-to-left pointer} \]

```c
int partition(itemType a[], int l, int r)
{
    int i, j; itemType v;
    v = a[r]; i = l-1; j = r;
    for (;;)
    {
        while (less(a[++i], v)) ;
        while (less(v, a[--j]))
            if (j == l) break;
        if (i >= j) break;
        exch(a[i], a[j]);
    }
    exch(a[i], a[r]);
    return i;
}
```

Issues

* stop pointers on keys equal to \( v \)?
* sentinels or explicit tests for array bounds?
* details of pointer crossing
Quicksort implementation

```c
quicksort(int a[], int l, int r)
{
    int i;
    if (r > l)
    {
        i = partition(l, r);
        quicksort(a, l, i-1);
        quicksort(a, i+1, r);
    }
}
```

Running time proportional to $N \times \log N$
running time for $N = 100,000$
    about .4 seconds
how long for $N = 1$ million ?
    slightly more than 10 times (about 5 secs)

Novices beware: could be slow for some inputs
[stay tuned, COS 226]
int partition(Item a[], int l, int r)
{
    int i = l-1, j = r;
    Item v = a[r];
    for (; i < r - 1; i++)
    {
        while (less(a[++i], v));
        while (less(v, a[--j]))
        {
            if (j == l) break;
            if (i >= j) break;
            exch(a[i], a[j]);
        }
        exch(a[i], a[r]);
    }
    return i;
}
void quicksort(Item a[], int l, int r)
{
    int i;
    if (r <= l) return;
    i = partition(a, l, r);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
}

Striking feature: NO huge numbers
Good algorithms are *more powerful* than supercomputers

Ex: assume that
home PC executes \(10^7\) comparisons/second
supercomputer does \(10^{12}\) comparisons/second

Running time estimates

<table>
<thead>
<tr>
<th></th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insertion sort</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>home PC</td>
<td>instant</td>
<td>1 day</td>
<td>3100 years</td>
</tr>
<tr>
<td>supercomputer</td>
<td>instant</td>
<td>1 sec</td>
<td>1.6 weeks</td>
</tr>
</tbody>
</table>

| **Quicksort**        |          |         |          |
| home PC              | instant  | 2.8 sec | 1 hour   |
| supercomputer        | instant  | instant | instant  |

Implementations and analysis validate each other
Further refinements possible
design-implement-experiment-analyze cycle
Computational complexity

Framework to study efficiency of algorithms
machine models, average case, worst-case
Upper bound: algorithm to solve the problem
Lower bound: proof that no alg can do better
Optimal algorithm: lower bound = upper bound
UPPER BOUND for sorting
mergesort algorithm
divide-and-conquer method
* sort two halves
* merge them to make sorted whole
always takes time proportional to $N \log N$
quicksort is usually faster
mergesort is guaranteed not to be slow
LOWER BOUND for sorting
THM: All algorithms use $> N \log N$ comparisons
Proof sketch:
$N!$ different situations
$\lg(N!)$ comparisons to separate them
$\lg(N!) > N \lg N$ [Stirling's formula]

Caveats
worst- or average-case may be unrealistic
costs ignored in analysis may dominate
machine model may be restrictive
Complexity studies provide
starting point for practical implementations
indication of approaches to be avoided
Some Important (But Hard) Problems

* TRAVELING SALESPERSON
  A salesperson needs to visit N cities. Find a route that minimizes travel distance.

* SCHEDULING
  A set of jobs of varying length need to be done on two identical machines before a certain deadline. Can the jobs be arranged so that the deadline is met?

* SEQUENCING
  A set of four-character fragments have been obtained by breaking up a long string into overlapping pieces. Can the fragments be reconstituted into the long string?

* SATISFIABILITY
  Is there a way to assign truth values to a given logical formula that makes it true?
Properties of Algorithms

Step-by-step set of instructions that can be applied in the same way to all instances of a problem.

* **FINITE**
  guaranteed to terminate
* **DETERMINISTIC**
  always produces the same answer given the same inputs

Algorithms are problem-solving methods suitable for use on computers. Example: multiplication.

A given problem can be solved by many different algorithms, but some algorithms are far more efficient than others.

**EFFICIENT:**
  "polynomial" time (ex: \( N^2 \)) for all inputs

**INEFFICIENT:**
  "exponential" time (ex: \( 10^N \)) for some inputs
Properties of Computers

Modern computers have varying characteristics

1970’s “mainframe”

1980’s “personal computer”

1990’s “microprocessor”

Supercomputer

Networks of computers

From a theoretical standpoint, they’re all the same!

1930’s Turing Machine

For example, none of the machines can solve the traveling salesperson problem for 1000 cities.
Some Numbers

$10^8$  PC instructions/second

$10^{12}$  supercomputer instructions/second

$10^9$  seconds/year

$10^{13}$  age of universe in years (estimated)

$10^{79}$  number of electrons in the universe (estimated)

Exponential growth dwarfs technological change:

Suppose each electron in the universe had the computing power of today’s supercomputers. If they worked together for the estimated life of the universe, they couldn't solve the traveling salesperson problem on 1000 cities.

.  1000  300  79  13  9  12
.  2  >  10  >> 10  * 10  * 10  * 10
Polynomial Time Algorithms

P:
The set of all problems solvable by deterministic algorithms in polynomial time.

Covers virtually all programs running on actual computers.
(Doesn't matter which computer.)

NP:
The set of all problems solvable by non-deterministic algorithms in polynomial time.

For a problem in NP,
a machine can efficiently VERIFY that a given solution is correct.

If a machine can guess (and is lucky), it can solve a problem in NP quickly.
Actual computers can simulate Lucky Guessing, in exponential time, by trying every possibility.
The Main Question

Is P=NP?

If not, then some problem must be
    in NP
    but not in P

(every problem in P is in NP).

Nondeterminism (Lucky Guessing) seems
    powerful, but no one has been able to PROVE
    that it helps for any particular problem.

Many important problems are in NP, but no
    efficient solutions for them have been found.
    (Only exponential algorithms are known).

Should we abandon hope of
    finding efficient solutions?
NP-Completeness

NP-Complete
A problem with the property that
if it can be solved efficiently, then P=NP.
(Lucky Guessing doesn’t help.)

For specific problems A and B, we can often show:
If A can be solved efficiently, then so can B.
Thus, if B is NP-complete, so is A.

Thousands of problems have been shown to be
NP-complete (including traveling salesperson) in this way.

If any one of these important problems can be solved efficiently, they all can.
(Moreover, so can any problem in NP).

But how was the first problem shown to be NP-complete?
SATISFIABILITY is NP-complete.

Outline of proof:

Non-deterministic Turing machines can solve problems in NP.

Logical formulas can describe Turing machines including non-deterministic ones.

Given any problem in NP establish a correspondence with some instance of SATISFIABILITY

SATISFIABILITY solution gives simulation of Turing machine solving the corresponding problem.

If SATISFIABILITY can be solved quickly, then so can any problem in NP.

Any problem reducible to SATISFIABILITY is NP-complete, etc.
Implications of NP-Completeness

Either:
Conventional machines can do as well as machines capable of Lucky Guessing, but we don't know how to make them do so. (P=NP)

Or:
Lucky Guessing DOES help, but is a fiction on conventional machines, since none of the NP-complete problems can be solved in polynomial time. (P!=NP)

Not many people believe that P=NP
...but it's possible.

Proof that a problem is NP-complete is usually taken as a signal to abandon hope of finding an efficient solution.
Coping With NP-Completeness

* Hope that the worst case doesn't occur  
  (try to simulate Lucky Guessing)

* Change the problem  
  (try for an approximate solution)

* Exploit NP-completeness  
  (example: cryptography)

* Keep trying to prove that P=NP!
Mechanical Calculators

Abacus (~1000 BC)

Slide rule (Napier, 1617)

Adding machine (Pascal, 1642)

Jacquard loom
  punched card control

Difference engine (Babbage, 1820)

Tabulating machines (Hollerith, 1890)
  punched card readers
  application: census

Telephone circuits
  relay-based
  counting
  memory
  switching

Desk calculators

Cryptanalysis/ballistics
Charles Babbage

Conceived the first
automatic digital computer

Built “difference engine” (1820)
mechanical calculator
special-purpose (evaluate polynomials)

Designed “analytical engine” (1833)
involved features of modern machines
“store” (memory unit)
“mill” (arithmetic unit)
“control” (instructions on punch cards)

Sponsored by Lady Ada Lovelace
daughter of Lord Byron
first computer programmer?

Idea from Jacquard loom?

“Father of modern computing?”
machine was impossible to build
work forgotten for a century

Other “notable contributions”
helped develop British postal system
invented the cowcatcher
Evolution of modern machine

Turing Bombe (WW-II)

Mark I (Aiken, 1944)
  "relay" computer/calculator

ENIAC (Mauchly/Eckert, 1945)
  "electronic" computer
  18,000 electron tubes
  100 multiplies/second
  plugboard "programming"

von Neumann
  stored program computer design

UNIVAC (1950)
  electronic, stored-program

  transistor (1947)
  core memory (1950s)
  integrated circuit (1960s)

microprocessor
VLSI microcomputer
Top ten programming languages

COBOL
FORTRAN
Algol
LISP
PL/I
Ada
Pascal
C
C++
Java
Applications

Military
  cryptography
  ballistics

Science/mathematics/engineering
  large-scale numerical calculations
  measurement/analysis
  simulation
  control

Commercial computing
  information processing
  word processing

“Personal” computing
  computer in every home?

Entertainment
Sample future applications

Put knowledge online

Human genome project

Medical technology
   "X-ray" vision

Simulation
   materials/cars/planes
   earth/biosphere
   space/matter
   economic systems
   brain

"paintbrush", musical instrument

Communications
   teleconferencing
   video on demand

Commercial/business systems

Artificial Intelligence
   vision/language understanding
   robotic control systems
   learning/problem-solving
Present and future computer systems

Only the numbers change!

SUPERCOMPUTER

"roomful of computer"
thousands of processors
$10^{12}$ instructions/second
$10^{12}$ bytes RAM
10 machines / $50$ million each

FILE SERVER

1000 users
$10^{12}$ bytes online storage
10,000 machines / $50$ thousand each

PERSONAL COMPUTER

"personal productivity tool"
word processor, database, graphics
C, Java, Maple, etc., etc.
$10^{8}$ instructions/second
$10^{8}$ bytes RAM
10 million machines / $3000$ each

NETWORK COMPUTER

"X-term for the millenium"
runs Java
$10^{9}$ instructions/second
$10^{7}$ bytes RAM
100 million (?) / $200$ each
Persistent Ideas

on-off switch
layers of abstraction
von Neumann machine
procedural languages
algorithms/complexity
keyboard/display
recursive programs and structures
formal languages/automata
memory hierarchy
printed page
symbolic mathematics
modelling physical objects
networks
timesharing/parallelism
computability/NP-completeness