1 CS1003/1004: Intro to CS, Spring 2004

Lecture #1: Introduction

Janak J Parekh
janak@cs.columbia.edu

2 What is this class?

- An introduction to Computer Science
- Two *required* components:
  - Weekly lecture covering the *theory* behind CS, common to both languages
  - Weekly lab covering a programming language, different one for each language
  - "Guinea pig" format
- Prerequisites: basic computer skills
- Which language is "better"?

3 Basic information

- Instructor: Janak J Parekh (janak@cs.columbia.edu)
  - Call me Janak, please
  - 9th year at Columbia (in various capacities)
  - OH: to be finalized once we get all our TAs
- Class website: http://www.cs.columbia.edu/~janak/cs10034
  - Make sure to check it regularly
  - Still setting up webboard and other sections…

4 Lab information

- C lab taught by TAs Suhit Gupta (suhit@cs.columbia.edu) and Java lab by Maryam Kamvar (mkamvar@cs.columbia.edu)
- Please register by end-of-week if possible
  - Difficulty in scheduling labs: who has a problem?
- Exception for this first week *only*: **no labs this week**
  - Instead, UNIX tutorial in this room this Thursday, 11-12:15pm

5 Textbooks

- Multiple textbooks
  - Brookshear, "Computer Science: An Overview", 7th Ed. required for theory
  - Oualline, "Practical C Programming", 3rd Ed., required for C lab
  - Bishop, "Java Gently", 3rd Ed., required for Java lab
  - Everyone must buy two textbooks (sorry!)
- Books can be obtained from Papyrus, SW 114th & Broadway; Amazon links & ISBN on website

6 Course structure

- 6 homeworks, 25 points each = 150 points
  - Roughly every 2 weeks
- 50 point midterm, 100-point final (open-book)

- Class participation (see next slide)
- In other words, homeworks are most important component of class
    - Learning programming is useless unless you actually do it hands-on

7 Class participation and attendance

- Attendance is expected; participation is beneficial
    - I won't take attendance, but the TAs might informally
    - Participation is useful for your grade at the end of the semester…
- If you miss class and/or lab, you're expected to catch up
    - I'll post slides and reading assignments to the schedule page to help

8 Homeworks

- Will consist of written and programming parts
    - Programming part will be submitted online
    - Programming to be done on CUNIX (or at least tested there)
- Late policy: you are given 3 grace days during the semester
    - A late day is exactly 24 hours
    - Can use up to two on any individual homework
    - After late days used up, late submissions will *not be accepted*

9 Homework 0

- It's up
- Basically, get your CUNIX account and make sure you can log into it
    - See if you can compile code
- Not to be submitted
- Thursday tutorial will cover most of these topics

10 Cheating

- Plagiarism and cheating: unacceptable
    - You're expected to do homeworks *by yourself*
    - Rest assured I have electronic tools to catch plagiarizers
        - I had five students last semester
        - Renaming variables, etc. doesn't help
- Results: instant zero on assignment, likely referral to dean
    - Columbia takes dishonesty *very seriously*
    - I'd much rather you come to me or the TAs for help

11 Feedback

- This is a "guinea-pig" course: I'm open to suggestions
- I can't promise I'll make your dreams come true, but I will take any constructive feedback seriously
    - Not just template-speak: ask my students from last semester
- I'm here to help you succeed!

12 Poll time!

- School
    - CC: 6
    - SEAS: 60
    - GS: 4

    - Other: 7
- Year
    - Freshman: 15
    - Sophomore: 15
    - Junior: 5
    - Senior: 7
    - Masters or later: 6

13 Poll (II)

- Have you programmed before?
    - No: 50
    - Yes (BASIC, VB): 6
    - Yes (C, C++, C#, Java): 4
- Have you used…
    - UNIX: 7
    - Windows command prompt: 10
- You're taking this class…
    - Because you want to: 15
    - Because you have to: 40

14 What is Computer Science?

15 What is Computer Science?

- We ask Google: http://www.google.com/search?q=define:Computer+Science
- I like this one best: "The systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application."
    - "Information age": we're presented with tons of information, and need tools to help organize it and manipulate it.

16 Who cares?

- "I'm taking this class because I have to know how to write code."
- "I'm taking this class because my advisor said I have to and I need an A."
- Several reasons:
    - Rising importance of computers in the world (and for your job)
    - A good coder does *not* necessarily make a good programmer or good computer scientist
    - Learning a programming language doesn't necessarily make a good coder
    - Brainteasers…

17 So what are we going to do?

- Study *algorithms*
    - An algorithm is a "set of steps that defines how a task is performed"
    - Not necessarily as intuitive as you may think
- Study *programs/software*
    - A program is machine-compatible representation of an algorithm, written in a *programming language*
- Study (the basics of) *hardware:* how does the software run?

18 Abstraction

- While we're studying all this, maintain the fundamental principle of abstraction
- What is abstraction?
    - http://www.google.com/search?q=define:abstraction
    - "Abstraction means ignoring many details in order to focus on the most important elements of a problem."

    - At any given time, we focus on one aspect of a problem, and abstract away the details of others
    - Lets us build a "big picture" of Computer Science, brick by brick

19 Topics we'll cover

- We'll start with the basics you need to start programming: language basics, algorithm design
- Then, we'll take a bottom-up approach to the computer
    - How is information stored in hardware?
    - How is information manipulated in hardware?
    - How do you tell the hardware to manipulate information?
    - How do you run this software in a reasonable fashion on a hardware?
- Finally, we'll look at some interesting directions for Computer Science
    - AI: the "future"?
    - Computation theory: what makes a computer a computer from a theoretical perspective?

20 And in the labs…

- A pragmatic approach to learning the programming language of your choice
- I'll work hard to synchronize the two parts of the class, although they won't always cover the same topics
    - You're *not* going to write an operating system!

21 Let's start thinking…

- You've got a five quart jug, a three quart jug, and a lake. How do you come up with exactly a gallon of water?
    - This is (was?) a brainteaser asked at Microsoft interviews

22 How to get a quart

- I'll model this as (x,y) where x == # of quarts in five-quart jug, y == # of quarts in three-gallon jug
    - 1. Fill three: (0, 3)
    - 2. Move three to five: (3, 0)
    - 3. Fill three: (3, 3)
    - 4. Move (as much as possible) three to five: (5, 1)
    - 5. Dump five: (0, 1)
    - 6. Move three to five: (1, 0)
    - 7. Fill three: (1, 3)
    - 8. Move three to five: (4, 0)

23 Something more pragmatic, perhaps?

- Given a map of the NYC subway system, design an algorithm that finds the "optimal route" between two stations
    - OK, this is not *that* easy, and you're *not* going to know enough to do this in this class
    - But we can think about it conceptually: got any ideas?
    - http://www.mta.info/nyct/maps/submap.htm

24 OK, how about something simpler?

- Given 10 numbers, sort them
    - Easy, you say?
    - Sort 100 numbers
    - Sort 1,000 numbers
    - Do it fast

25 Being a good programmer…

- Takes more than knowing how to write code
- It takes the ability to take a problem and break it down into small enough steps to write code that solves it
- It takes the ability of knowing enough of the field (and the language) to know what a “step” is
- Hopefully, that’s what you’ll learn this Spring

26 Before we go any further…

- Let me prove that I, unlike most professors, know how to program
  - All of us know C and Java, so don’t hesitate to ask for help
- First program: always “Hello, world!”
- We’ll go through the details next week…
- I’ll put this code up; try running it for HW#0

27 Next class

- **NO LAB THIS WEEK!**
- Next class will be on Thursday, 1/20, 11am-12:15pm
  - UNIX tutorial

1 CS1003/1004: Intro to CS, Spring 2004

Lecture #2: Intro to UNIX

Janak J Parekh
janak@cs.columbia.edu

2 Administrivia

- Textbooks should now be available from Papyrus – has anyone tried to pick them up?
- Awaiting confirmation on increasing section 2 size for 1114
  - We'll probably move the room
- Please register!

3 A "Warning"

- I'm about to cover a lot of material in 75 minutes
- I don't expect you to get everything initially, but try and understand *the basics* of what's going on
- Stop me and ask questions!
  - Especially if I type something too quickly…

4 What is UNIX?

- UNIX was an operating system invented in AT&T/Bell Labs in the 70s
- Became extremely popular as it was easily adaptable to a variety of computing hardware, and because it supported multiuser/multitasking environments
- Who owns "UNIX" now?
- Linux is not UNIX -- but is *very* similar
  - SCO is just plain wrong, IMHO

5 Why do you need to know UNIX?

- Columbia's main computing cluster runs a version of UNIX
  - Sun's Solaris 9 == Solaris 2.9 == SunOS 5.9
- Provides an "equal" and robust environment for everyone to work in
- Useful for many engineering fields, or as a background for anyone interested in Computer Science
  - Resume material

6 Is UNIX user-friendly?

- No.
- Well, it's getting better, but for many years, UNIX was considered "hacker/programmer-friendly"
  - Simple example: commands are generally very short
- UNIX is heavily command-line driven
  - A "command-line" is a textual way of interacting with a computer, one line at a time
  - Windows has a command-line too: Start => Programs => Accessories => Command Prompt
  - Less intuitive, but very powerful

7 How do you log onto CUNIX?

- Through an AcIS Solaris-based machine
    - In particular, 251 Engineering Terrace: full graphical UNIX interface (known as X)
        - Requires extended account, unlike other AcIS labs
- Via a remote machine: use telnet or ssh (Secure SHell)
    - Advice: *Don't* use telnet – it's insecure, and AcIS will be turning it off
    - AcIS provides a free ssh client, TeraTerm – let's take a look…

8 Useful UNIX commands

- 1 **ls:** List files
- **mv:** move/rename files
- **cp:** copy files
- **rm:** remove files
- **cat:** print out a file
- **mkdir:** make directory
- **rmdir:** remove directory
- 2 **cd:** change directory
- **pwd:** print working directory
- **man:** manual page
- **gcc, javac:** compilers
- **emacs, pico, vi:** editors
- **more, less:** pagers
- **lpr:** print (in 251)

9 Directory structure

- Ever used Windows Explorer?
- A "/" is the *delimeter* to separate out parts of the *pathname*
    - Windows uses "\"…
    - Just "/" is the root: *no* drive letters in UNIX
- "..": parent directory
- All your files are in /{home}/*UNI/*
    - On CUNIX, not literally "home", some prefix
    - ~ or ~/cs10034 is easiest way to reference your "home"

10 UNIX environment

- You run in a "shell", typically bash
- "Settings" that apply when you're logged in
- PATH: where to look for programs to run (including the aforementioned UNIX utilities, which are in /usr/bin)
    - Sometimes, may need "./a.out", not "a.out"
- **set, export:** Lets you manipulate the environment
    - "export CLASSPATH=/home/jjp32/javacode"
    - Goes into "~/.profile" if you want it to be automatic
- Don't worry about this yet, just keep it in mind…

11 Pipes, redirection

- Lets you reroute output from one program to a file (redirection) or to another program (pipes)
- **ls > test.txt:** Puts list of files in test.txt
- **less < test.txt:** Cat's test.txt through a pager

- **ls | less:** Useful if you have a long list of files

12 Editors

- Pico: The "Pine Composer" – *very* easy to use, but very plain-jane
- Emacs: "Editor MACroS"
  - Extremely powerful
  - http://c2.com/cgi/wiki?EmacsStandsFor
  - I recommend this, especially "over" X – auto-indenting will save you *many* times over
- Vi: "Visual Interpreter"
  - Want to be l33ter than me? Learn this
- Windows tools, IDEs: you can use, but not supported

13 X

- The X Window System is the GUI for UNIX
- Invented at MIT in the 80s
  - X11 was released in the 90s
- Supports "remote displays" over the network
  - "X server" is the display: you can download one for Windows at http://www.cs.columbia.edu/crf/crf-guide/resources/software/xwin32.html
- Tip: Use the "X Forwarding" option in TeraTerm's ssh client, start up X server, have fun
- emacs is 100 times easier this way…

14 If you don't have broadband…?

- Various workarounds
  - Get broadband
  - Stay connected for long times, and don't use X
  - Use 251 ET
  - Set up a UNIX-like environment on Windows
- Windows has a command prompt called "cmd" (NT/2k/XP) or "command" (95/98/Me)
- For 1003: cygwin gives you a UNIX shell, gcc, ls, etc.
- For 1004: Java Development Kit from Sun gives you javac
- Emacs can be downloaded for free, too
- See the resources page tonight for links to the above

15 Transferring files

- Especially for those of you working from home, might want to copy files back and forth
- FTP: File Transfer Protocol
- AcIS provides WS_FTP for free
  - Insecure :-/
  - PuTTY has a free Secure FTP client, but it's command-line based; see resources

16 Other useful utilities

- finger, who, w: See who's logged in, get more info
- lookup: Columbia's white pages
  - Not everyone is listed though
- fortune: OK, not necessarily useful, but fun

17 Additional resources

- I know this tutorial was admittedly quick…
- Web-based tutorials on UNIX and emacs:
  - http://www.columbia.edu/acis/webdev/unix/index.html

  - http://www.columbia.edu/acis/publications/emacs.html
  - More links on Resources page
- AcIS will have hands-on training sessions in 252 ET
  - How many 1003 students interested? (Java by default)
  - I'll mail a list of the sessions
- Come see me or the TAs: we're happy to help
  - I'll try to have a TA hold office hours in 251 ET

1 CS1003/1004: Intro to CS, Spring 2004

Lecture #3: Intro to Programming Languages

Janak J Parekh
janak@cs.columbia.edu

2 Administrivia

- Buy those textbooks – the Papyrus guy is after me!
- Third TA
- Labs start this week
  - Section 2 for 1114 has been moved & increased to 40 students
    - Room is a little hard to get to – see instructions on the class website
  - Labs are more recitations than labs per se
  - Consolidation?
  - At least one set of OH in 251 ET
- Register for the webboard
- AcIS training sessions
- Office hours
- Who hasn't registered for a lab?

3 Agenda

- Finish up UNIX tutorial, talk about HW#0
- Segue into programming
  - What exactly does the code do, and why?
  - General programming concepts you need to know
- HW#1 to be released this week
  - Programming is *very* easy, and very short: more a piggyback off of HW#0 than anything else
  - Check the website
  - You've got plenty of time, so *relax*

4 UNIX redux

- filename~: not the same thing as ~/filename
  - The latter is a "backup" file generated by editors like emacs
- Files in UNIX are case-sensitive
  - HelloWorld.java vs. helloworld.java vs HELLOWORLD.java
- "cd" by itself is equivalent to "cd ~" or "cd ~/"
  - However, ~/ lets you reference files/directories *absolutely* as well, which cd doesn't

5 UNIX (II)

- Two sets of files: those on the server vs. on your computer
  - Use FTP to move things back and forth…
- Other questions from last time?

6 So, what to do for HW#0?

- *Not* freak out
- Let's do it right now, step by step
- *Please* ask me questions now if you don't get it…
- Steps:

  - Get HelloWorld.java or hello.c onto CUNIX account
  - Go into CUNIX and run compiler
  - Run the code
- What does the code mean?

7 What does the code mean?

1
```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println(“Hello world!”);
  }
}
```

2
```
#include <stdio.h>
int main() {
  printf(“Hello world!\n”);
}
```

8 Why do we program this way?

- A machine generally processes very primitive calculator-like instructions:
  - “Get first number from memory”
  - “Get second number from memory”
  - “Add the two numbers”
  - “Store the results back in memory”
- All of this is in binary code (machine language)
  - An “operation” might be 0111001010010101001001010100010101
  - We’ll learn how this works later
- In short: yuck!

9 One step up

- Instead of using hard-to-read machine language, use textual representations
  - LD R1, x (*load the value of X into R1 in the CPU*)
  - LD R2, y
  - ADD R0, R1, R2
  - etc.
- *Assembly* language: considered “second-level” language
- Still really annoying: what we want is “x + y”

10 3rd-generation languages

- Started in the 50s/60s with FORTRAN and COBOL
- Idea: take a higher-level description of what we want to do, and let the computer *translate* it into the machine language as specified before
- Called *compiler* because it might take a single high-level command, and compile a sequence of low-level commands
  - Input high-level language as text, store binary commands in *executable file*
- Alternative: *interpret* commands on the fly and issue low-level statements to the processor (BASIC does this)
- C is compiled; Java between compiled and interpreted

11 4th-generation languages

- Very high-level languages; historically intended for user-friendliness
- Many “application-specific” languages
  - Matlab might be construed as one
  - Rapid development tools (database languages, Visual Basic, etc.)

- Tends to do a lot of the work *itself*
- We'll focus on 3rd-generation languages in this course; skills can be used in 4GLs

12 Different kinds of 3GLs

- C and Java are *procedural* or *imperative* languages
  - You define *procedures*, or sets of steps, to solve
  - Java is also considered an *object-oriented* language
- Not the only way to program
  - Declarative programming: you declare "facts": Excel
  - Functional programming: you develop "functions", and then build them up; very similar to a set of equations
  - Won't look at these, although there is some conceptual overlap
- Object-oriented programming: model on top of the others that specify how to organize information and code; we'll talk about this later

13 Elements of procedural programming

- Procedure declaration
  - Mathematical function is a decent model, actually
  - What are the inputs?
  - What are the outputs?
- Declarative statements: define terminology to be used later in the program
- Imperative statements: actually perform actions related to what we want
- In C and Java, each declarative/imperative statement **must** end with a semicolon
- Comments: not actually processed; merely for human readability

14 General model of procedural programming

- Get some information from user
- Process the information
- Give the user some results
- How does Hello World follow this model?
  - Input: we don't need anything: we already know what we're going to output
  - Process: nothing to process, since we already know the output
  - Results: print out "Hello world!"
- Some other simple examples…

15 Compiling

- The compiler takes the source code you write in *text form* and produces binary output
- As it goes along, it checks your source for *syntax errors*
  - Errors may be cryptic at times
  - There are errors which the compiler won't be able to detect (*semantic errors*)
- If there are no errors, it spits output, and quits
- You can then run your program on the machine
  - For Java, must run through an *interpreter*
  - For C, it's machine code: just run it!

1

CS1003/1004: Intro to CS, Spring 2004

Lecture #4: Language concepts, data storage

Janak J Parekh
janak@cs.columbia.edu

2

Administrivia

- HW#1 is out!
  - I hope you're checking the website frequently
  - Should know everything for the HW this week
  - Programming is about 5 lines of code, so don't worry too much
- Fourth TA: Rob Tobkes
  - Info on website
  - We now have office hours 5 days a week
- Labs update
  - How'd your first lab go?
  - *This week only:* Suhit's combining Thursday C labs to see what works best
- Register for the webboard, or else!
- Put books on reserve?

3

Agenda

- Finish up language intro
- Start data representation concepts
- Hopefully everything you need for the theory part of HW#1
  - If not, I'll trim the HW#1 theory a little bit
- Some overlap with labs…

4

Variables

- Very often, we want to store information from user as *data*
- We can do so by *declaring variables*
  - In C or Java, a *declarative statement*
    "datatype variablename [= value];", e.g.
    "int i = 5;"
  - Conceptually similar to a mathematical variable, but we try to be more precise and assign the variable a *data type*
- We can then assign *values* to these variables
  - From user input
  - As the result of some computation
  - Even random numbers

5

What data types?

- Lots; you'll see some of them in the labs
- Some basics…
  - int = Integer, generally between -2 billion and positive 2 billion
  - double = Floating-point (i.e., flexible number of decimal places), roughly between -10^{308} and 10^{308} (although not an infinite number of decimals!)
  - char = Character (such as 'a')
  - Strings (i.e., words, sentences or arbitrary alphanumeric data) are complicated ☹
- We'll talk about storage shortly…

6

And more…

- We can even declare *arrays* of variables
  - Since we're not going to have 50,000 declarations at the beginning of every piece of code
  - "int j[10];" in C, "int j[] = new int[10];" in Java
  - Access array by *index*, e.g., "j[5] = 15;"
  - Note array is homogeneous, not heterogeneous
- Can get much more complicated by this, but let's not worry about that yet

7 Constants and literals

- We don't need to declare variables for everything; as we saw, we can just *literally* put numbers in place when we want to do things
  - e.g., print the sum of 10 and 15
- We can also declare that certain variables are *constants* for sanity's sake
  - "const double Pi = 3.141592654" in C
  - "final double Pi = 3.141592654" in Java

8 Assignments

- Once we've *declared* our variables, we might want to assign them values
  - x = 5;
- Can do this at declaration-time, too
  - int x = 5;
- Key concept: the above two statements are not functionally equivalent!
- *Operators* commonly used in assignments
  - * for multiply, + for add, - for subtract…
  - Operator precedence applies: use parentheses!

9 Comments

- As your code becomes more complex, you'll want to document it a little
- In C and Java, can use "/* comment */" notation
  - Can be multiple lines
- In Java, can also use "// comment" notation
  - Single-line only
  - Sometimes works in C too, but depends on age of compiler

10 Control statements

- We generally want to adjust the behavior of our program based on the situation
  - Options in a menu: *if* the user clicks Save, *then* save the file. *Else if* the user clicks Exit, *then* Exit. And so on…
- In older programming languages, "goto" would exist
  - Considered bad form nowadays, because it can lead to very confusing code
- Instead, the *if-then-else* construct is used
  - if(something) do something
    else if(something else) do something else
    else do a generic thing
- Generally, control statement itself doesn't need a semicolon

11 What's "something"?

- A *boolean condition*
- That is, if the test clause evaluates to *true*, then the corresponding code is executed
- Use curly braces ({,}) to "group together" code to be executed
- if(numcredits > 20) {
    printf("You're insane!");
  }

12 What is a boolean value?

- In Java, there is a data type called *boolean*
  - Can be assigned “true” or “false”
- In C, no such datatype; you can use an int to represent it
  - 0 is false, any nonzero value is true (1 is common)
  - Can “create” a boolean datatype, much later in the semester
- Why 0 and 1?
  - Three more slides…

13 What are boolean operators?

- A *logic operator* that takes one or two operands and produces a boolean result
- For numbers:
  - Equals: ==
  - Greater than: >
  - Less than: <
- Extremely important: “=” is not “==”
  - “=” is an **assignment operator**, while “==” is a **boolean test**
  - C programmers: *you will get burned by this at least once in your life*
  - Java programmers: the compiler will usually warn you

14 Combine boolean values?

- AND: &&
  - Only true if both operands are true
- OR: ||
  - Only false if both operands are false
- NOT: !
  - Takes single operand and reverses it
- We can draw “truth tables” for each of these
- Let’s do a few examples…

15 Loops

- Instead of doing something *once*, can we do something *many times* until a boolean condition is satisfied?
  - Yes, we can
  - while(something is true) do something
  - Will keep on running (potentially forever)
  - How can we make an infinite loop (not that we’d want to)?
  - How can we make our loops non-infinite?
- for statement: more complex notation for loops
  - In labs…
- *Iteration* is the fancy term for such repetition

16 How is this information represented in the machine?

- *Bit* (binary digit): either 0 or 1
  - Why?
- What can we do with bits?
  - Combine them together into larger values
  - Base 2 representation of numbers…
    - Converting from decimal to binary: divide by 2 repeatedly and keep the remainder
    - Converting from binary to decimal: multiply the i^{th} digit by 2^i (with i starting at 0 for the ones’ digit)

17 Binary representation, cont’d.

- We can also represent characters (in general) as a binary sequence
  - ASCII: American Standard Code for Information Interchange
  - Originally used 7 bits to represent a single character
  - Now, 8 bits used == *byte* in most computers today
  - Google for “ASCII table”
- Finally, we can apply *logic* operators to bit values
  - AND, OR, NOT, XOR are the four basics
    - Why XOR?
  - We’ve already seen the first two…

18 AND and OR

19 NOT and XOR

20 Logic diagrams

- Use those four building blocks to build increasingly complex logic operators, and ultimately devices
- Example: how would we diagram a AND b AND c?

21 Next time

- Finish up data storage
- Start talking about understanding algorithms using all our newfound information

1 CS1003/1004: Intro to CS, Spring 2004

Lecture #5: Data storage, algorithms

Janak J Parekh
janak@cs.columbia.edu

2 Administrivia

- HW#1 is due Thursday
- HW#2 will come out at about the same time
- TA office hour changes
  - Check the website and webboard on a weekly basis
- Another UNIX tutorial session via the ACM
  - Not hands-on; more of a lecture style
  - Wednesday 7:30pm, 252 ET
- Is the board readable?

3 Agenda

- Finish up data representation
  - I'm going to skip flip-flops and two's complement until later in the semester, when it fits better
- Start algorithms discussion

4 Why do you care about bits?

- These are the basic building-blocks of a computer
- It turns out you can build *everything* up from those four primitive operations!
- Bit and logic constructs pervade throughout a programming language as well
  - Logic constructs are fundamental to programming

5 Some bits-and-bytes trivia

- 8 bits typically == 1 byte
- Blocks of memory done in powers of 2
- 2^{10} bytes == 1024 bytes == 1 kilobyte
- 2^{20} bytes == 1,048,576 bytes == 1 megabyte
- 2^{30} bytes == 1 gigabyte
- Confusion with metric terms
- Several different kinds of memory
  - RAM – Random Access Memory – very fast
  - Hard disks, CDs, tapes – mass storage systems – generally slower

6 Algorithm basics

- An algorithm is "an ordered set of unambiguous, executable steps".
  - Ordered – does not imply "followed in order"
  - Executable – each step must be doable
  - Unambiguous – during execution, information in the state of the process must be sufficient to determine, uniquely and completely, the actions required by each step
  - Implies that the algorithm *terminates* with a result
    - The "halting problem"

7 Why do we care?

- Applies to real-world circumstances as well
  - Every activity of the human mind actually the result of an algorithm execution?
- Difference: we understand the real-world context
  - Once we understand the digital context, programming ultimately becomes easy
- Challenge: representing an algorithm
  - English is lousy for this
  - A major challenge in software design

8 So how do we represent algorithms?

- Several key building blocks
- Primitives (+, -, etc.)
  - Insufficient by itself for "higher-level" code – too repetitious, much like assembly
- Higher-order language constructs
  - *Assignment* (a = b + 5)
  - *Conditionals* (if (a > 10)…)
  - *Loops* (while (a < 20)…)
  - *Procedures* (c = random())

9 What's psuedocode?

- A way of approximating the syntax of real code without getting lost in the syntactic details
- In essence, a cross between English and code
- Useful when trying to design an algorithm on paper
- In this class, I'll generally avoid psuedocode except when necessary
- You're welcome to use the book's model or my model

10 Procedures?

- I've dealt with this implicitly, but let's be more formal
- How does printf(…) or System.out.println(…) work?
  - Someone else has written the code to handle printing
  - These *procedures* may take *parameters* and may *return* a result
  - Note – many parameters, single result!
- Called *functions* in C, *methods* in Java

11 Why procedures?

- Code reuse
  - If we design a mathematical operation, we don't want to have to write it out repeatedly
- Code organization
  - Lets us "segment" the code to make it more readable and manageable
- Enables abstraction
  - Worry about the details of a particular task in its own procedure, not elsewhere

12 Declaring a procedure in C or Java

- Basic concept: just name one
  - Three parts: procedure name, return value's datatype, and argument list
  - Argument list is a pair of datatype and *variable name*
  - Why no name for the return value?
- Let's write a very simple example: finding the average of two numbers

13 Organizing code

- What does our main() function do, then?
- For any non-trivial program, generally main() is used to set up and control the program, and then all the handling is done in subsidiary functions
  - In C, order of functions may matter
  - In Java, *constructors* are also used for setup purposes
- This way, we avoid a 5,000-line main()
- Learning optimal organizing takes time and experience

14 How do we come up with algorithms?

- An imprecise science at best: problem-solving
  - Understand the problem
  - Get an idea of how/which algorithm might solve the problem
  - Formulate the algorithm and represent as a program
  - Evaluate the program for accuracy and potential to solve other problems
- This is not much help, is it?

15 "Get a foot in the door"

- Try doing the first (few) step(s) by hand
  - Look at what you had to do to accomplish it
  - See if you can reapply this to continue solving the problem
- Reapply another solution
- Stepwise refinement
  - Look at the problem from a very high level
  - Break it down repeatedly into smaller pieces, until we get a set of algorithmic steps

16 Iterative structures

- Very often, we need to *repeat* steps in order to solve a problem
- A number of basic methodologies that do precisely this
  - Sequential search algorithm
  - Loop-based control
  - Sorting
- Warning: need to keep track of *boundary conditions*

17 Let's try some simple examples

1. Print out the first *n* numbers, and keep a running total
2. Print out the first *n* Fibonacci numbers
3. Write a function that calculates *x^y* (i.e., raise x to the y power)
4. Reverse a list (array) of numbers

18 Next time

- Look at another approach to algorithm problem-solving
- Discuss how to compare algorithms and their efficiency

1

CS1003/1004:
Intro to CS, Spring 2004

Lecture #6: Algorithms II

Janak J Parekh
janak@cs.columbia.edu

2

Administrivia

- HW#2 is out
  - You *really* should start earlier for this one…
- HW#1 being graded
  - Most people seemed to do well on the programs
  - If you couldn't do the HW#1 programming, come see me and let's straighten it out – future homeworks will only be harder
  - Questions? Feedback?
- Yet another ACM UNIX session this Wednesday (more advanced stuff), 7:30, 252 ET

3

Agenda

- Sidebar: good homework practices
- Continue algorithms discussion

4

Homework notes

- As I suggest, make sure you know what you want to do first, *conceptually*, before programming it
- How to debug your code?
  - First - recognize if your error is syntax or semantics
  - Learn how to understand the compiler's error messages
  - Try going through the code by hand and make sure it makes sense
  - Put *debugging statements* in your code
  - If you are truly stuck, go to a TA's office hours or email them a *detailed* bugreport
  - Don't send code!

5

Homework notes (II)

- Commenting your code
  - I didn't require it for HW#1, but I want you to start for HW#2
  - /* … */ and // conventions
  - What to comment?
    - Put your name and a brief description at the top of your source file
    - Put a comment before things that are non-obvious
    - Put a comment before non-obvious functions
    - Wherever else you feel appropriate
- Look at my examples…

6

Review of last class

- Strategies with coming up with algorithms…
  - "Get foot in the door": try to get an intuitive grasp on the problem first, conceptually
  - Stepwise refinement: take the big picture and break into smaller pieces
  - Determine if there are any iterative structures to be implemented
  - Keep boundary conditions in mind!

7 Iterative structures, cont'd.

- Two more types of loop constructs
- for: useful for situations where we're doing a loop N times
  - for(i=0; i < 10; i++) { … } runs exactly 10 times
  - Three parts: initialize, condition, increment
  - for(; i < 10;) { … } == while(i < 10) { … }
  - Java: can put declaration inside for loop, e.g.,
    for(int i=0; i < 10; i++) { … }

8 Iterative structures, cont'd.

- do-while: almost the same as while, but it does one run *first*
  - do { … } while (0>1); will run how many times?
  - Less used
- Another paradigm: use the *break* keyword
  - Will break out of loop, sometimes useful if you find you don't need to run through every step
  - while(true) { … break; … } is sometimes used – not usually good form

9 Let's revisit our examples

1. Print out the first *n* numbers, and keep a running total… *using a for loop*
2. Print out the first *n* Fibonacci numbers
3. Write a function that calculates *x^y* (i.e., raise x to the y power)
4. Reverse a list (array) of numbers

10 Here's another way to look at repetition

- fib(n) = fib(n-1) + fib(n-2), right?
- We can actually encode that in a computer
  - *Recursion:* Define a solution in terms of a smaller version of itself
  - Must have *stopping* (base) case(s)
  - What's the base case for the above recursion?
- How about doing *x^y* using recursion?

11 Another recursive example

- Binary search: works for a sorted list of information
- Basic idea: pick the middle element
  - If that's what we're looking for, done
  - If it's larger, recursively search the "top half"
  - Otherwise, recursively search the "bottom half"
  - If we're stuck with an empty list, we failed

12 HW#2

- Asks you to check a *palindrome*
- I'm not going to do the homework for you, but let's think, conceptually, what needs to be done…

13 Next time

- Finish up intro to algorithms

1 CS1003/1004: Intro to CS, Spring 2004

Lecture #7: Algorithms III

Janak J Parekh
janak@cs.columbia.edu

2 Administrivia

- HW#2 due this week
  - I'll cover running times today
- HW#1 being returned between last week and this week
  - We'll coordinate returns better in the future
- Midterm in two weeks
  - Format of the midterm
  - I'll post a list of topics next week
  - Extra review session?

3 Agenda

- Finish algorithms discussion (for now)

4 Here's another way to look at repetition

- fib(n) = fib(n-1) + fib(n-2), right?
- We can actually encode that in a computer
  - *Recursion:* Define a solution in terms of a smaller version of itself
  - Must have *stopping* (base) case(s)
  - What's the base case for the above recursion?
- How about doing *x^y* using recursion?

5 Other recursive examples

- Power (x^y)
- Binary search
- Palindrome checking
- Most iterative structures can be done recursively, and vice-versa

6 Algorithm efficiency

- Often, there's multiple ways to implement an algorithm
- How to characterize if one's better or not?
- Two primary considerations:
  - How *fast* does an algorithm run?
  - How much *memory* does an algorithm take?
- Let's focus on the first one for now

7 Our multiple Fibonacci algorithms

- Do they run at the same speed?
- Let's try fib(10)… then 20… then 40

- Hmm, why do they differ?
- And can we classify this difference

8 How fast does an algorithm run?

- Let’s first think of it in the context of *steps*
- How long might a linear search take through a list of N elements?
- Canonical way to characterize this is to use “big-Oh” notation
    - Key insight: we’re interested in orders of magnitude, not constants
    - Strangely, book uses big-Theta notation, which is less used except when doing more formalized analysis

9 Big-Oh notation

- Basic intuition:
    - Find the number of steps in terms of *n* or other variables
    - Drop any constants or additive lower-order terms
    - Put a O( ) around the result
- Let’s look at the previous algorithms we discussed today and see what their big-Oh complexity is…

10 Other algorithms?

1. An algorithm to compute n! – recursively
2. Sort the contents of an array
    - I don’t like insertion sort – let’s do bubble sort

- We’ll continue to do more “interesting” algorithms as the semester proceeds

11 Next time

- Continue algorithms

1 CS1003/1004: Intro to CS, Spring 2004

Lecture #8: Algorithms IV
Janak J Parekh
janak@cs.columbia.edu

2 Administrivia

- HW#2 due now
    - Won't be returned before midterm, so I'll release solutions
- HW#3 out
    - All programming
- I'm teaching C lab this week
- Midterm next Tuesday
    - Topics list posted
    - Extra review session?

3 Agenda

- One more recursive example
- Talk about one more class of algorithms: *sorting*
- Spend some more time on big-Oh notation
- Midterm review
    - More midterm review in labs…

4 Recursion, redux

- Idea: instead of using explicit loops, cast problem in terms of itself
- *Base case(s)* and *recursive case*
- How can we compute n! recursively?
- I won't make you design a recursion on the exam, but you should be able to recognize one

5 Sorting

- Common problem: given data, sort it in some fashion
- Most common-type is *comparison-based sort*
- Can you come up with way to sort information?
- Many different kinds; we'll look at two today
    - Bubble sort
    - Insertion sort
- Let's make this interesting…

6 Big-Oh notation, redux

- Basic intuition:
    - Find the number of steps in terms of *n* or other variables
    - Drop any constants or additive lower-order terms
    - Put a O( ) around the result
    - Common: O(1), O(log N), O(N), O(N^2), O(2^N)
- What's the complexity of the algorithms we just talked about?

7 Next time

- Midterm
- Then break! ☺
- Then HW3 is due… ☹

1 CS1003/1004:
Intro to CS, Spring 2004

Lecture #9: Midterm review, data structures

Janak J Parekh
janak@cs.columbia.edu

2 Administrivia

- HW#3 due now
- HW#4 out today
  - Less programming, more written
  - Some programming based on HW#3; I'll release solutions you can work off of if you want
- Midterms returned now

3 Midterm statistics

4 How I grade?

- Grades added up *at end of semester* and then "scaled" appropriately
- Median grade in the class is borderline B/B+
- Remember, class participation helps
- Marked improvement also helps
- Come talk to me if you have any questions

5 Midterm answers

- Part 1
  - CS1003: F, T, F, T, F
  - CS1004: F, F, T, T, F
  - I allowed partial credit, though
- Part 2, Q1
  - Algorithm finds *top two* numbers
  - Removing italics => val2 no longer is the second-highest number
  - O(n) algorithm

6 Midterm answers cont'd.

- Part 2, Q2
  - 46 and 23
  - Dropping the last bit does integer division by two
- Part 2, Q3 – runs *9* times (i=1 through i=9)

```
int i = 1;
while(i < 10) {
  System.out.println(i); or printf(“%d\n”, i);
  i++;
}
```

7 Midterm answers cont'd.

- Part 3: Note that prime #s start at 2!

```
int nextPrime = 2, numPrimes = 0;
```

```
while(numPrimes < n) {
  if(isPrime(nextPrime)) {
    print(nextPrime);
    numPrimes++;
  }
  nextPrime++;
}
```

8 Why HW#3?

- I know it was a large programming assignment, but it was a necessary one
- In essence, summarized the "first half" of the semester
- You need these skills under your belt for the rest of the semester
- If you didn't quite finish, take a look at solutions, come to office hours, etc. and *make sure you understand*

9 Bubble sort, reviewed

```
for(i=alength - 1; i > 0; i--) {
  for(j = 0; j < i; j++) {
    if(a[j] > a[j+1]) {
      int temp = a[j];
      a[j] = a[j+1];
      a[j+1] = temp;
    }
  }
}
```

- Why is this $O(n^2)$?

10 Insertion sort

- Similar to bubble sort; *slightly* more efficient
- Principle: consider the left side the "sorted" side, and the right side the "unsorted" side
- Successively insert the "next unsorted" element into position into the "sorted" side
- Applets demoing this and Bubble sort: http://home.janak.net/cs3134/lafore-applets/Chap03/
- You can use either sort…

11 Data structures

- We've been referring to this informally, but now let's be precise
- A computer's memory is a large open space, and we can organize information in it
- A *data structure* is an organized entity in this memory space
- The most primitive data structures: *primitive types*

12 Primitive types

- int, char, double, etc.
- Occupy a well-known amount of memory
  - For 32-bit machines, an char takes *1 byte*, an int takes *4 bytes*, a double takes *8 bytes*
  - *Not always the case*, but enough for this class
- The variable refers to that block of memory in its entirety
  - Can't typically store decimal places inside an int; "won't fit"
- But what if we want something more complicated?

13 Arrays

- I've arbitrarily defined these as a block of memory divided into cells
- To be more precise, an array is a *static* structure in memory
    - Memory is organized "contiguously" when you define an array
        - 10 integers => 10 * 4 => 40 bytes on a 32-bit machine
    - The variable referring to the array actually just points to the *beginning* of the appropriate memory location

14 Arrays (2)

- The programming language then does some math when you use [ ] to access an index in that array…
    - An array of integers, length 10 is at memory location "4000".
    - How many bytes is this array in total?
    - What's the position of the 5th integer?
    - Rationale for 0-based makes a little more sense

15 More generally…

- For primitive datatypes (int, char, etc.), the variable refers to that entity *in its entirety*
- But whenever we work with a more complex data structure than just a primitive datatype, our variable will "point" to the beginning of the structure
    - Known as a *pointer* (C) or a *reference* (Java)
- The programming language then decides what part of the memory starting at the variable you're working with

16 Strings

- Strings are an interesting case
- In C, Strings are just arrays, and we treat them as blocks of memory of predefined size
- In Java, Strings are *dynamic*, and can vary in length
    - We'll get into more technical details later
- Here's why doing == with Strings doesn't work, though…

17 Custom data types

- Wouldn't it be nice for HW#3 to have a single "entity" to refer to bank account, so we can have an array of *bank accounts* instead of two separate arrays?
- We can declare such a *structure* (C) or *object* (Java)
    - We'll set it up so that it contains a String and a double
    - We then access *components* of that "bank account"
- This week's lab will start with the basics on how to do exactly this

18 How complicated?

- Data structures & types can be almost as complicated as you want
- You can nest complex data structures
    - For example, a bank account can contain an array of dependents
    - You can have an array of bank accounts in a Branch
    - You can have an array of Branches in a BankInstitution
    - And so on…
- How can we organize all this stuff!?
    - Take CS3134, and you'll learn all the details. Here's a few.

19 "List" data abstraction

- The most common way to organize things is in a list

    - An array is one type of a list – it's *static* sizewise; "contiguous list"
  - What are basic *conceptual* operations on a list?
  - Can we organize lists in any different fashion?

20 Next time

  - Continue discussion on data structures

1

CS1003/1004: Intro to CS, Spring 2004

Lecture #10: Data structures II

Janak J Parekh
janak@cs.columbia.edu

2

Administrivia

- HW#4 due next Tuesday
- I'll be in Seattle next week; Suhit will lecture in place of me
  - He's eminently qualified for the next topic
- 1 point on midterm problem 3…

3

Custom data types

- Wouldn't it be nice for HW#3 to have a single "entity" to refer to bank account, so we can have an array of *bank accounts* instead of two separate arrays?
- We can declare such a *structure* (C) or *object* (Java)
  - We'll set it up so that it contains a String and a double
  - We then access *components* of that "bank account"
- You should be learning language-specific skills for this now

4

How complicated?

- Data structures & types can be almost as complicated as you want
- You can nest complex data structures
  - For example, a bank account can contain an array of dependents
  - You can have an array of bank accounts in a Branch
  - You can have an array of Branches in a BankInstitution
  - And so on…
- How can we organize all this stuff!?
  - Take CS3134, and you'll learn all the details. Here's a few.
  - You won't have to worry about the implementation details – we're focusing only on the basic concepts

5

"List" data abstraction

- The most common way to organize things is in a list
  - An array is one type of a list – it's *static* sizewise; "contiguous list"
- What are basic *conceptual* operations on a list?
- How do these conceptual operations work with an array?
- Can we organize lists in any different fashion?

6

Linked List

- Idea: instead of allocating *one* block of memory and dividing it into individual cells, create lots of individual scattered cells and connect them together in one long chain
- Advantages:
  - Infinite-length – just allocate another block
  - Easy to insert or remove an element in the middle
- Disadvantages:
  - Lots of memory management

7 Stacks and Queues

- Variation on lists to support specific problems
- Stacks follow a *LIFO* policy (last-in, first-out)
  - "Push" and "pop" operations
- Queues follow a *FIFO* policy (first-in, first-out)
  - Enqueue, dequeue
- Both have numerous applications in computing
  - Stacks used to keep track of procedure calls
  - Queues used for print queues

8 Trees

- Instead of just a linear data structure, why can't we have something more flexible?
- Trees are called such because they have nodes that are arranged into a hierarchy with a *root*, *leaves*, and *children*
- Most popular kind of tree is a *binary* tree, where every node has two children
- Binary *search* trees provide faster ways to search of information: O(log n) for insert, remove, search

9 Yes, this is a whirlwind tour

- Data Structures, W3134, covers all of these in much greater detail, including implementation
- Just make sure you understand the concepts and the basic algorithms involved with them
- Brookshear has a decent discussion of these

10 Next time

- Suhit will teach you guys the basics of a computer (i.e., computer architecture)

1 CS1003/1004: Intro to CS, Spring 2004

Lecture #11: Computer Architecture

Suhit Gupta
suhit@cs.columbia.edu

2 Administrivia

- HW#4 due today
- Janak's office hours today
  - Rob and I will be available

- Reiteration of plagiarism policy
  - VERY SERIOUS
  - I recommend sending email to Janak

3 Computer Architecture

- In this class, you are studying software
- But how does this relate to the hardware in your machine
- Two aspects
  - At the "macro" level, how is the computer organized
  - At the "micro" level, what is the architecture of each component

4 The Macro - The Computer

5 The Micro – The Motherboard & The Processor

6 Computer Architecture in Software Perspective

7 The CPU

- CPU = Central Processing Unit
  - consists of two parts
    - ALU – Arithmetic Logic Unit
    - Control Unit
- The CPU contains talks to the machine memory (RAM) and the system cache, but it also has internal memory called registers

8 The CPU-Memory Relationship & Hierarchy

9 Chip Architecture (MIPS)

10 Instruction Set

- So how does software run on a machine
- The CPU only understand machine instructions and computes 1's and 0's
  - Therefore, something has to convert it to machine language – enter "compiler"
    - The compiler converts high level code into machine code (this is why you have machine specific compilers)
  - RISC – Reduced Instruction Set Computer
    - machines are efficient and fast
    - limited
    - code density is awful
    - examples: MIPS, DLX, (ARM/Thumb)
  - CISC – Complex Instruction Set Computer
    - complex and slower (to some extent)
    - code density is excellent
    - examples: Intel, PowerPC, (ARM/Thumb)

11

Machine Language

- Machine language – series of instructions that have been converted from some higher level language
  - it is something that the processor understands.
- machine instruction

- Machine instruction consists of opcode (operation code) and a number of operand fields

12

Machine Language example

- In C/Java, a simple piece of code to search for k in an array would look like

```
while (array[i] == k)
      i++;
```

- In MIPS assembly language, it would look like

```
Loop:      mult   $9, $19, $10        ; Initialize i
           lw     $8, Sstart($9)      ; Get value of array[i]
           bne    $8, $21, Exit       ; check if it is equal to k
           add    $19, $19, #1        ; i++
           j      Loop                ; back into the loop
Exit:
```

13

Also included in the architecture

- Program counter
  - contains the address of the next instruction
- The machine cycle

14

Back to the Chip Architecture

15

Pipelining (using DLX assembly)

- Blocks of code are typically large
  - One cannot execute each instruction, one at a time
  - Therefore, execute them together?
  - Pipeline them

```
LOOP: LW      R8,   0(R2)
      ADD     R10, R6,   R8
      ADDI    R2,   R10, #4
      SW      R10, 0(R2)
      ADDI    R3,   R3,   #4
      LW      R1,   100(R3)
      LW      R12, 100(R1)
      BGTZ    R12, LOOP
LOOP: LW      R8,   0(R2)
      ADD     R10, R6,   R8
      ADDI    R2,   R10, #4
```

16

17

Communication via controllers

- Communication between a computer and other devices is typically handled through an intermediary device called a *controller*
- A controller converts messages and data back and forth for compatibility
- Each controller is assigned unique addresses
  - Set of addresses assigned is called a port
- Memory mapped I/O
- Direct Memory Access (DMA)
  - wonderful for performance
- von Neumann bottleneck
  - CPU and controllers, both trying to access the machine bus

18

Multiprocessor machines

- Pipelining can be viewed as the first step towards supporting multiple processors (parallel processing)

- Common pitfall: mutiple processors is different from multiple processes
- Common design pitfall: throw lots of workers at a task and it will get done faster
  - Works with extreme delicacy in Software Engineering
  - Works better in hardware but makes design much harder

19 Advanced concepts

- SISD – Single Instruction, Single Data
  - typical of what we have seen so far
- MIMD – Multiple Instruction, Multiple Data
  - in multiple processor machines, one processor can store the program information, then call on another processor to complete it
- SIMD – Single Instruction, Multiple Data
  - typically VLIW machines (Very Long Instruction Word)

20 Final thoughts and the next class…

- I cannot stress this the plagiarism policy more firmly than I already have

- Operating systems & networks
  - Read Chapter 3 of the Brookshear book

1 CS1003/1004: Intro to CS, Spring 2004

Lecture #12: OS & Networks

Janak J Parekh
janak@cs.columbia.edu

2 Administrivia

- Three weeks left in the semester!
- HW#5 due next Tuesday
  - If you have not started already… you'd better start today
  - Don't expect to write the programming up over a weekend
- Thanks to Suhit for teaching last week
  - How was he? ;-)

3 The big picture

4 The big picture (II)

- Given hardware and compiled (machine) code, you can run it directly, but that's a huge hassle
  - What if you want to run multiple programs?
  - If so, how do we share resources between programs?
  - How do we let the user manipulate various programs?
  - How do we let *multiple users* manipulate various programs?
- Solution: employ a special piece of software that allows multiple user applications/tasks to cooperate

5 History of operating systems

- *Batch processing:* back in the single-task days, people would submit jobs to the computer for the entire company, and wait in line for their job to be done
  - Used a *queue* abstraction to handle the job list
  - No interactivity – submit job, wait for results
  - Very cumbersome for iterative development
- Interactive processing
  - Allow the user to interact
  - Still had to wait for your shot to use the computer
  - Anyone remember DOS?
- Modern OSes *multitask*

6 Operating systems

- Considered *system software*, as compared to *application software*
  - The latter run as *processes* alongside an OS
- Two major components:
  - A *kernel*, which handles resource management, multitasking, etc. in the background;
  - A *shell*, which provides a user frontend to the operating system

7 Kernels

- Several important components
  - *Device drivers:* used to enable the OS to communicate with computer hardware
    - Device drivers *abstract* the hardware away from the OS, so that you can "plug-in" new drivers
  - *Memory manager:* Keeps track of computer's memory allocation per process; also supports *virtual memory*, which enables

    the use of hard disks as additional memory
  - *Scheduler:* Control what tasks are running on the processor at any given time
  - *Network stack:* Provides networking facilities

8 The Linux kernel

- Popular learning kernel, since it's open source
- You can grab your own copy from www.kernel.org, if you want to take a look
- A Linux operating system distribution (like Red Hat) consists of the *Linux kernel* and a bunch of tools (including GNU tools)
- Here's the directory structure of the kernel…

9

10 Multitasking

- Given multiple *processes*, coordinate them so that they can run concurrently
- Well, not concurrently – the CPU handles a fixed number of instructions at any given time
  - Instead, *timeslice*, so that each process does a little work at a time, and keep on switching
  - Operating system keeps separate register sets, etc. for each application, and magically handles them cleanly for you
  - "Virtual machine": As an application designer, you *feel* like you have control over the machine, but the OS is actually managing many such processes

11 Multitasking (II)

12 How do *you* multitask in UNIX?

- The "&" operator
  - "emacs &" starts up emacs as a *background process*
  - Lets you continue to use the shell while running emacs in its own window
  - "jobs" lists the currently running jobs in the background
- Or… multiple ssh sessions
- The machine is actually handling all of these user sessions in parallel as collections of processes
  - UNIX is *multiuser*, unlike older client versions of Windows

13 Multiuser and other trivia

- By being multiuser, UNIX must worry about user accounts, passwords, and permissions
  - *root:* administrative UNIX account (like Windows "Administrator" user)
- "w" or "finger" will list the currently logged-in users on the current machine
  - Note that CUNIX is a *cluster* of machines, not just one machine
- "ps" lists the processes on a machine
  - "ps auxw" (Linux/BSD) or "ps –ef" (Solaris/SysV)
  - top lists most active processes on a machine
- "kill" kills a process

14 Process competition?

- What if two different processes need to access the same resource?
  - In the old days, if two programs want to print, you'd get a printout that was a mix of both
  - Now, a *print spooler* coordinates output and keeps them separate
  - The OS is responsible for handling such *race conditions* between processes

15 Process competition (II)

- More complicated resource contention requires *locking*; concept is similar to the barriers at a

train track crossing
  - Semaphores == fancy locks
- Avoid *deadlock:*

16 Networks

- Now that we've discussed all the pieces on *one* computer, let's talk about networking computers together
- More and more computing solutions are *distributed* across networks
- Several different kinds:
  - LAN (Local Area Network)
  - WAN (Wide Area Network)

17 LANs

- Most common LAN architecture today is Ethernet
- 10BASE-T/100BASE-T Ethernet use telephone-like wire to network computers together
  - Very cheap, and popular ("CAT 5" wiring)
- *Topology:* how to organize these networks?
  - Typically a hierarchical star topology nowadays
  - Columbia's network is a hybrid of Ethernet and fiber

18 WANs

- Typically collections of LANs, with high-speed telecommunications links connecting them together
  - POTS (plain old telephone system): typically < 56kbps
  - DSL/cable: typically 128kbps-1.5Mbps
  - T1: 1.544Mbps
  - T3: 45Mbps
  - OC3: 155Mbps
  - OC12: 622Mbps
- Columbia has an OC3 to the commodity Internet
  - not enough…

19 The Internet

20 The Internet

- A very, *very* large WAN
- http://research.lumeta.com/ches/map/gallery/index.html
  - *Extremely* complicated
  - "The Internet has a diameter of 10,000 pookies"
- Active research as how to accurately map Internet topography
  - We just had a Ph.D. student come yesterday as a faculty candidate talk on this very topic

21 So how does the Internet work?

- On top of a series of *network protocols* that define how computers should talk to each other
- Internet Protocol (IP) is the most important
  - Current one (IPv4) was made over 20 years ago(!)
  - http://www.ietf.org/rfc/rfc0791.txt
  - Next version is IPv6: "coming soon"
- Describes how computers should be *addressed*, how to *route* between networks, and how to carry data

22 IP addressing

- IPv4: “dotted-quad notation”
    - Each machine has an address of the form xxx.yyy.zzz.www
    - Many “restricted” addresses
    - DNS (domain name service) maps a name to an IP address
    - chambers.psl.cs.columbia.edu ➔ 128.59.14.155
- LANs typically have contiguous IP addresses
    - Columbia (wired): 128.59.*.*
    - Columbia (wireless): 160.39.*.*
    - We’re getting slowly more fragmented
- *Routers* “route” packets between one LAN to another based on addresses and a “routing table”

23 IP “packets”

- A *packet* is a bag of data, typically up to 1500 bytes
- Contains some *headers* specifying things like source and destination, and some *data*
- The Internet is a “packet-switched” network
- TCP (Transmission Control Protocol) is one protocol that takes large amount of data to be sent and breaks them up into these small packets
- TCP/IP – the most common combination (RFC 793)
- I can take a look at the packets if I’m bored…

24 What services run on the Internet?

- E-mail: specified by its own protocols
    - SMTP (RFC 821, 2821) – Specifies how to transfer email from a source to a destination via a chain of mail servers
    - POP3/IMAP are simply *retrieval* protocols to retrieve your mail from a mailbox
- Web: two main standards
    - HTTP: Hypertext Transfer Protocol (RFC 2616)
    - HTML: Hypertext Markup Language
- Both work over TCP/IP
    - “Stacking” protocols on top of each other
    - *Port* abstraction to separate services over TCP/IP

25 Other services

- Telnet: simple text over TCP/IP
    - In fact, I can telnet to an HTTP server and talk HTTP or SMTP if I know how to
- FTP: File Transfer Protocol
- ssh: like telnet, but encrypted for security’s sake
    - I can actually read the data typed over telnet or ftp using tcpdump… if I’m root or have control over a switch
- Others?
    - kazaa, AIM, MSN, you name it
    - Once you learn more, you can make your own

26 So how do you stay secure?

- Effective password management
    - Change your passwords every so often
    - Don’t use your last name as the password
- Use secure protocols
    - These use *encryption*, which makes it difficult for a third-party
    - SSL, ssh are two of several out there
- Don’t run random programs on your computer
    - Viruses and spyware can do network traffic communication behind your back, and convey your own data to other parties

27 What does this mean for you?

- OSes and networks are the context of all the work we do with computers nowadays
- If you program in the future, you’ll likely have to interact with both in a more involved form
- Both C and Java have ways of communicating with the operating system and with other computers on LANs and the Internet, so you can write your own Kazaa’s or webbrowsers…

28 Next time

- In labs:
    - C – more pointers and structs
    - Java – basic graphics programming
- ***Make sure to come to us with questions this week***
- Lecture: basic AI concepts

1 CS1003/1004: Intro to CS, Spring 2004

Lecture #13: Networks, AI

Janak J Parekh
janak@cs.columbia.edu

2 Administrivia

- HW#5 due today
- HW#6 out tonight
  - Homework topics feedback?
  - It's not always easy coming up with "interesting stuff" that isn't very hard.
- Maryam will be out *next* week
  - William will be teaching her lectures
  - There may be some OH rescheduling, so be sure to check the webboard
- Grades
- Review session?

3 IP addressing

- IPv4: "dotted-quad notation"
  - Each machine has an address of the form xxx.yyy.zzz.www
  - Many "restricted" addresses
  - DNS (domain name service) maps a name to an IP address
  - chambers.psl.cs.columbia.edu ➔ 128.59.14.155
- LANs typically have contiguous IP addresses
  - Columbia (wired): 128.59.*.*
  - Columbia (wireless): 160.39.*.*
  - We're getting slowly more fragmented
- *Routers* "route" packets between one LAN to another based on addresses and a "routing table"

4 IP "packets"

- A *packet* is a bag of data, typically up to 1500 bytes
- Contains some *headers* specifying things like source and destination, and some *data*
- The Internet is a "packet-switched" network
- TCP (Transmission Control Protocol) is one protocol that takes large amount of data to be sent and breaks them up into these small packets
- TCP/IP – the most common combination (RFC 793)
- I can take a look at the packets if I'm bored…

5 What services run on the Internet?

- E-mail: specified by its own protocols
  - SMTP (RFC 821, 2821) – Specifies how to transfer email from a source to a destination via a chain of mail servers
  - POP3/IMAP are simply *retrieval* protocols to retrieve your mail from a mailbox
- Web: two main standards
  - HTTP: Hypertext Transfer Protocol (RFC 2616)
  - HTML: Hypertext Markup Language
- Both work over TCP/IP
  - "Stacking" protocols on top of each other
  - *Port* abstraction to separate services over TCP/IP

6 Other services

- Telnet: simple text over TCP/IP
    - In fact, I can telnet to an HTTP server and talk HTTP or SMTP if I know how to
- FTP: File Transfer Protocol
- ssh: like telnet, but encrypted for security's sake
    - I can actually read the data typed over telnet or ftp using tcpdump… if I'm root or have control over a switch
- Others?
    - kazaa, IRC, AIM, MSN, you name it
    - Worms
    - Once you learn more, you can make your own

7 So how do you stay secure?

- Effective password management
    - Change your passwords every so often
    - Don't use your last name as the password
- Use secure protocols
    - These use *encryption*, which makes it difficult for a third-party
    - SSL, ssh are two of several out there
- Don't run random programs on your computer
    - Viruses and spyware can do network traffic communication behind your back, and convey your own data to other parties

8 What does this mean for you?

- OSes and networks are the context of all the work we do with computers nowadays
- If you program in the future, you'll likely have to interact with both in a more involved form
- Both C and Java have ways of communicating with the operating system and with other computers on LANs and the Internet, so you can write your own Kazaa's or webbrowsers…

9 Transition…

- We've already talked about…
    - Hardware basics
    - Software basics
    - Systems and networks
    - How to build solutions from these (albeit simple)
- This and the next lecture talk about more open-ended areas of Computer Science
    - But still very legitimate!

10 Artificial Intelligence

- Perhaps one of the most misunderstood Computer Science concepts
- "… to develop machines that communicate with their environments through traditionally human sensors means and proceed intelligently without human intervention."
- In other words:
    - Algorithms to understand human communication
    - Algorithms to process information unattended
- Once something "works", it's no longer "AI"
    - Voice recognition is here, and it works (mostly)

11 What's an AI?

- In order to accomplish the task, do we just use a clever combination standard computing algorithms (performance), or do we actually try to "model" the mind (simulation)?
- Is intelligence measured by the ability to win (at a game) or to be humanlike?

    - Turing test
    - Turing supposed that by the year 2000, machines would have a 30% chance of passing a 5-minute Turing test
    - DOCTOR/ELIZA: free copy in emacs!

12 Various AI methodologies

- Reasoning/production systems
- Neural networks
- Genetic algorithms
- Natural language processing
- Robotics, vision
- Databases/expert systems

13 Reasoning

- Common problem domain – the 8-puzzle
- There are 181,440 different configurations of the 8-puzzle
- Given a random configuration, can we compute the moves necessary to restore to this state?

14 A large search problem

15 "Production system"

- Consists of three things:
    1. A number of *states*
    2. A number of *productions* or *rules* to transition between states
    3. A *control system* to decide which rule to follow
- Given these elements, the problem reduces to a *search problem*
- One way of modeling this is a *search tree*, consisting of part of the state graph

16 Search tree for 9-puzzle

- This is just a partial search tree
- Represents one initial configuration
- Goal: to traverse the tree quickly enough and find the correct state
- Problem: tree can be very "wide"

17 Search tree for Tic-Tac-Toe

- Again, partial search tree
- User might be the first move, followed by a computer move, etc.
- Goal: find a *winning* state
- Problem reduced to a data structure and a set of *search algorithms*
- Still many choices…

18 Search strategies

- *Breadth-first*
    - Look at the first row, then the second row, then the third row…
- *Depth-first*
    - Go all the way to one leaf, then backtrack and resume
- *Heuristic*
    - Have a special piece of code that "tells" you a preferred choice
    - A directed search – not always foolproof, but reduces amount of nodes searched
    - For 8-puzzle: "# of tiles *out of place*" – take move that minimizes this value

19

Neural networks

- Idea modeled after neurons
- Given some inputs and a configuration, the neuron *fires* with the appropriate stimuli
- Neurons may "learn" which stimuli to fire on

20

Artificial neural networks

- Difference: we use numbers, not electrical impulses
- "Compute effective unit" uses weights w_x
- Goal: arrange a network of these that produces the result that we want, and adjust the weights so it gives the correct answer

21

Artificial neural networks (II)

- Challenge: Given such networks, we don't want to adjust the weights manually
- A technique called *backpropagation* allows the machine to be given "training data", and it adjusts its weights to match the desired output
- Example: face, voice recognition

22

Genetic Algorithms

- Have programs *evolve*; mix-and-match them to produce the best result
  - Common in building game players: mix-and-match players to produce desirable output
- Need a very focused language that you can "mix-and-match"
- Generally a very slow process to evolve

23

Natural Language Processing

- Syntactic analysis
  - Apply grammar rules
  - For example, identify the subject of the sentence "Mary gave John a birthday card."
- Semantic analysis
  - Identify the semantic role of each word, i.e., action, agent of action, object of action
- Contextual analysis
  - "I ate a bag of chips."
- Applications
  - Information retrieval and information extraction
  - Particularly important for web-based applications

24

Next time

- In labs:
  - C – File I/O
  - Java – GUI-based event programming
- Last lecture: finish up AI, computation theory

1 CS1003/1004: Intro to CS, Spring 2004

Lecture #14: AI, Computation Theory, The End

Janak J Parekh
janak@cs.columbia.edu

2 Administrivia

- HW#6 due next Wednesday – work on it!
- Maryam out this week
  - William will be teaching her lectures and covering Thursday's OH
  - We'll cover her OH next Monday as well
- OH requests next week?
- Review sessions – Tues. and Thurs.
  - Room to be finalized; will send email
  - Got preferences?
- Grades are up, please check them out ASAP

3 AI continued: Robotics/vision

- Historically focused on mechanical and electrical engineering aspects
- We can already do set tasks, but what about modifications?
  - Objects on a conveyor belt at irregular intervals/orientation
  - Navigate around a room with obstructions
- Need to take images of scenes, compute boundaries, detemine paths
- Goal: autonomous robots

4 Database/expert systems

- Context drives a huge problem: how to encode context and knowledge that the human mind possesses, and retrieve said information?
- "Associative memory systems"
- Web search is just a start – just keyword-based searching so far, not semantic-based searching
- Expert systems: encode domain-specific knowledge to help solve problems

5 Weak vs. Strong AI

- All of these applications are essentially *weak*: we tell the computer what to do, and we solve problems
  - Not really "AI", per se – useful solutions to solve real-world problems
- *Is* Strong AI, i.e., sentience/consciousness, possible?
  - If so, we're still quite a long way away
  - On the other hand, there's the Turing test…

6 So… what can't computers do?

- (Or, can we summarize what *can* they do?)
- Given all that we've learned this semester, it's actually pretty hard to characterize
- Focus of *computation theory* is to determine what is computable and what is not
  - Computable implies functions whose output values can be determined algorithmically from their input

values
- So, what’s an example of a noncomputable function?

7 Formalizing computability

- Several popular ways
  - (Finite) state machines
  - Turing machines
- State machines are a sort of like a flowchart
  - One starts at a “start state”, goal is to get to the “end” or “goal” state
  - State *transitions* specify what to do based on initial input
  - States represent the “current” computer’s state
  - Simple example: build a state machine to match the string “Hello!”
  - Problem: intermediate storage?

8 Turing machine

- A state machine on steroids
- Idea: not only do we have *state*, but we have *storage*
- Alan Turing modeled the storage as a “paper tape” in 1936
- The tape is manipulated by a read/write head that can move left and right one space

9 Simple Turing example

- Add one to a number already encoded on tape
- We encode it as a binary number, and surround it with the start/end states (“*”)
- Let’s do this on the board…

10 So why bother with Turing?

- Church-Turing thesis: the set of Turing functions is the *same* as the set of functions that are computable in general!
  - Although some may look *really* awkward in a Turing machine
- Widely accepted by computer scientists today
- A language is *Turing-complete* if it can encode all that a Turing machine can do
  - Both C and Java are Turing-complete

11 Noncomputability, redux

- So, noncomputable functions can’t be modeled as a Turing machine
- How do we demonstrate?
  - Not that trivial, beyond scope of class
- Most famous noncomputable function: *Will a specified program halt?*

12 The “halting problem”

- In short, we *cannot* compute whether or not a computer program written in a Turing-complete programming language will run to completion or not!
  - Note that the *program itself* is “input” into this noncomputable function (e.g., willHalt(…))
- Informal proof is in book; strictly optional (but you may find it interesting)
  - Bare-Bones also optional

13 Classes of computable functions

- We typically break them down by the time they take to run; here are some typical values that we’ve seen:

14 “Bad” computable functions

- Those that, for *any implementation*, take exponential time
- For sufficient *n*, these problems take so long to run that no matter how fast your computer is, it’ll still take practically forever
- What’s scary, though, is that there is (currently) *no way of proving* that there is *no faster way* of computing it
  - While recursive Fibonacci is bad, iterative is not

15 So…

- We call such functions for which we know no better way to be “nondeterministic polynomial”, or NP
  - Typically exponential
- We care because lots of useful problems fall into this category

16 How does one “prove” NP?

- You show that one non-polynomial problem reduces into another non-polynomial problem
  - **NP-complete** problem
  - Can’t do for all NP problems, but for many of them
  - It’s a “weak” proof: if one were to demonstrate that there exists a polynomial-time algorithm for *at least one* non-polynomial problem, *all* NP problems are automatically “P”
  - Prove “P=NP”: Insta-Nobel Prize. Guaranteed!

17 In fact, NP is “useful”

- Public-key encryption (e.g., SSL/ssh) largely works on the fact that decrypting an encrypted message takes an extraordinarily long time
  - Details beyond scope of class
- If someone were to prove that P=NP, many of today’s encryption algorithms would have to be thrown out the window
- Fortunately, no one has come close to proving it
- But no one has come close to proving the opposite either

18 So where do we go from here?

- Most computer scientists (except great theoreticians) focus on making new computable algorithms, hopefully in polynomial time
- With the knowledge you’ve learned in this class, you have the pieces to go ahead and build such algorithms, and code them
- Remaining CS classes introduce advanced concepts, but they still boil down to the same thing

19 Next time

- No next time ☹
- In labs:
  - C – Modularity, Makefiles
  - Java – packaging, Java API
- Final two weeks from today
- Wait! We’re not finished

20 Final

- Structure: very similar to midterm, about 50% longer – so you shouldn’t need all three hours

- I *will* put up a reading list by the end of this week that will cover section reading in great detail
  - Will tweak slides to remove stuff we didn't get to in class…
- Review sessions next week: they'll be open-ended, so bring questions!

21 Feedback

- This class, as I said at the beginning, is experimental
- *Please* fill out the SEAS Oracle survey
  - http://oracle.seas.columbia.edu
  - You can win an iPod!
- But let's also discuss the class now
  - I'm writing a report, and what you tell me can help
  - Final exam bonus anonymous survey?

22 Thank you!

- You guys have been a great audience.
- I hope you found this class rewarding.
- Good luck with the rest of your Computer Science mini-careers!
  - And with finals
- Don't forget review sessions next week