# CS1003/1004: Intro to CS, Spring 2004

Lecture #5: Data storage, algorithms

Janak J Parekh
janak@cs.columbia.edu

## 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?

## 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

## 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

## 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

## 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"

## 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

## 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())

## 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

## 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

## 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

## 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

## 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

## 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?

## "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

## 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*

## 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

## Next time

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