computing: nature, power and limits—robotics applications (cis1.0)
fall 2006—lecture # B.2
monday 18-sep-2006

today

- finish HTML concepts from last class
- introduce algorithmic thinking
- dance steps example
- reading: Reed chapter 8

what is an **algorithm**?

- "a step-by-step sequence of instructions for carrying out some task"
- examples of algorithms outside of computing:
  - cooking recipes
  - dance steps
  - proofs (mathematical or logical)
  - solutions to mathematical problems
- in computing, algorithms are synonymous with *problem solving*
- *How to Solve It*, by George Polya
  1. understand the problem
  2. devise a plan
  3. carry out your plan
  4. examine the solution
- example: find the oldest person in the class (besides me)

analysis of algorithms

- often, there is more than one way to solve a problem, i.e., there exists more than one algorithm for addressing any task
- some algorithms are better than others
- which *features* of the algorithm are important?
  - speed (number of steps)
  - memory (size of work space; how much scrap paper do you need?)
  - complexity (can others understand it?)
  - parallelism (can you do more than one step at once?)
- **Big-Oh** notation
  - $O(N)$ means solution time is proportional to the size of the problem (N)
  - $O(log_2 N)$ means solution time is proportional to $log_2 N$
  - see examples in Reed page 142

classic algorithm example: search

- *sequential* search
- *binary* search
- search the Manhattan phone book for "Al Pacino":
  - how many *comparisons* do you have to make in order to find the entry you are looking for?
  - *equality* versus *relativity*—which will tell you more? which will help you solve the problem more efficiently?
  - can you take advantage of the fact that the phone book is in *sorted* order? (i.e., an "ordered list")
  - what would happen to your algorithm if the phone book were in random order?

algorithms and programming

- programming languages provide a level of abstraction that is more understandable to humans than binary machine language ($0's$ and $1's$)
- *assembly languages* (in the early 1950's) provided abbreviations for machine language instructions (like MOV, ADD, STO)

- *high-level languages* (introduced in the late 1950's) provided more "programmer-friendly" ways for humans to write computer code (e.g., FORTRAN, LISP)
- program translation
  - translates assembly or high-level languages into binary machine language
  - two methods:
    * *interpretation*:
      reads and translates statements one at a time; doesn't optimize across an entire program; doesn't store executable statements—just runs them; error checking only happens at "run time" run-time can be slow, but there's no "compile time"
    * *compilation*:
      reads and translates entire program, and stores result as an executable file; can optimize; can perform "compile time" error checking; run-time is fast, but there is "compile time"
- concepts:
  - *compile-time (noun)*:
    the process of compiling a program from an assembly or high-level language into binary machine language and storing it on the computer's hard disk
  - vs *compile time (adj noun)*:
    the amount of time it takes a *compiler* to translate (or "compile") a program
  - *run-time (noun)*:
    the process of executing a compiled, stored program
  - vs *run time (adj noun)*:
    the amount of time it takes a program to run
    this is where *Big-Oh* comes in
  - *errors*:
    can be found at compile-time and at run-time
  - *error checking*:
    is done at compile-time