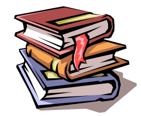
Lecture R1: Course Review

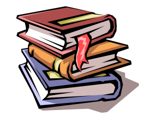
What You've Learned (A Lot!)

Programming.

- Basic skills are universal (C, Java, PostScript, Maple, Perl, TeX).
- Key abstractions:
  - structured programming: for, while, if, function call
  - data structures: array, struct, linked list, stack, queue, tree
  - pointer, recursion, divide-and-conquer
- Can address important problems without relying on pre-packaged solutions.



COS 217



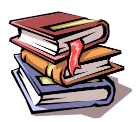
COS 226

What You've Learned (A Lot!)

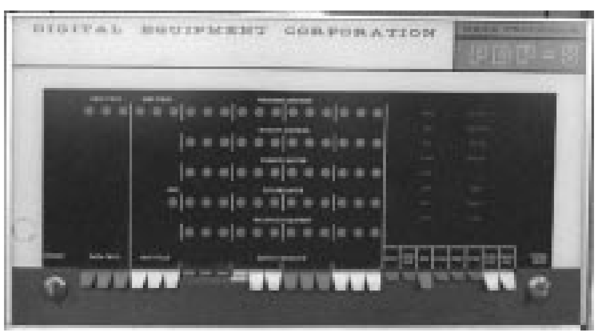
Programming.

The TOY machine.

- Bridge between C language and hardware.
- Machine language programming (0's and 1's).
- von Neumann architecture.
- Building a TOY machine from logic gates.



COS 306



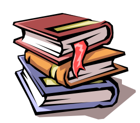
What You've Learned (A Lot!)

Programming.

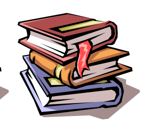
The TOY machine.

Theory of computation.

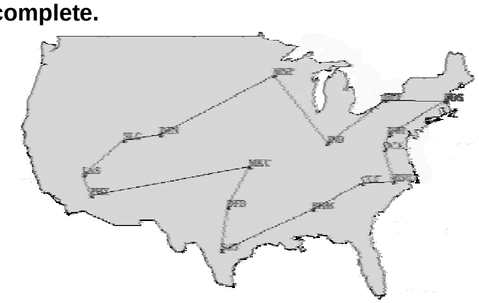
- Use formal language to model computation.
- Use abstract machines to strip away inessential details.
- Computability: all machines have limitations.
- Church-Turing thesis: Turing machine is all-powerful.
- Algorithms: polynomial vs. exponential.
- Problem classes: P, NP, NP-complete.



COS 423



COS 487



What Is Computer Science?

What is computer science?

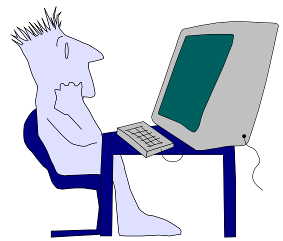
1. The science of manipulation "information."
2. Designing and building systems that do (1).

Why we learn CS.

- Appreciate underlying principles.
- Understand fundamental limitations.

An example: ~~Lecture I1: LFBSR   TOY machine~~ → ????

- How to make a simple machine.
- What can we do with it? What can't do with it?
- How fast can we do it?
- Science behind it.



Course Themes

Layers of Abstraction:

- Building a computer program.
  - divide program into small independent functions
  - ADT
- Building a computer.
  - transistors ⇒ gates ⇒ maj, odd ⇒ adder ⇒ ALU
  - ALU, register file, decoder, multiplexer ⇒ TOY machine
- Formal languages.
  - abstraction to describe computation
- Models of computation.
  - abstract machines, complexity classes

Course Themes

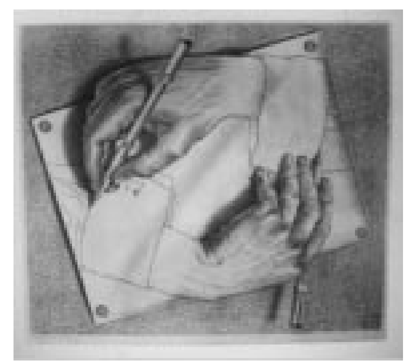
Tradeoffs:

- Time vs. space.
  - arrays, linked lists, BST
- Program generality vs. simplicity.
- Correct answer vs. time.
  - TSP brute force vs. heuristics
  - NP-completeness
- New machine vs. new idea.
  - machine cost $$$ and makes "everything" run incrementally faster
  - new ideas can enable new research and technology
- Expressiveness of language vs. ability to compile.
  - English is expressive: difficult for a computer to parse
  - C uses context-free grammar: easy to parse

Course Themes

Self reference:

- Recursion.
  - function that calls itself
- Linked list, tree.
  - self-referential data structures
- Fractal.
  - Mandelbrot set, H-tree pattern
- Sequential circuit.
  - feedback loop
- von Neumann architecture.
  - data and instruction stored in same main memory
- Universal Turing machine.
  - can simulate any machine including itself
- Undecidable problem.
  - key step in Halting proof was feeding one program itself as input

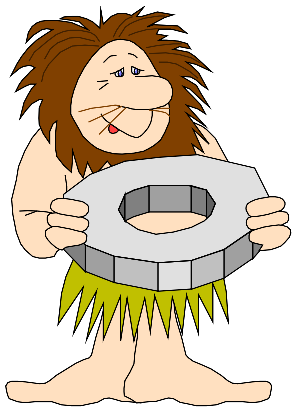


M.C. Escher

Course Themes

Reuse (don't reinvent the wheel):

- Loop.
  - let computer repeat code
- Program.
  - borrow similar program as template
- Function.
  - reuse code
- Circuit.
  - reuse primitive components
- Divide-and-conquer.
  - reuse ideas recursively
- ADT.
  - build general purpose libraries



What To Do When You Face a New Problem?

What primitive objects are important?

- Numbers, files, pictures, text, programs, strings, matrices?
- 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.

What To Do When You Face a New Problem?

Will I be doing something like this *frequently*?

- Is it worthwhile to learn a new tool?
- Is it worthwhile to *create* a new tool?

Has *someone else* done something like this?

- May be some code laying around to reuse.

Will someone else be doing something like this in the future?

- Document the code?
- Make it portable?

"Whenever we think a problem is simple, it turns out to be complicated. Fortunately, whenever we think it to be complicated, it turns out to be simple."

What To Do When You Face a New Problem?

Will I be doing something like this *frequently*?

- Is it worthwhile to learn a new tool?
- Is it worthwhile to *create* a new tool?

Has *someone else* done something like this?

- May be some code laying around to reuse.

Will someone else be doing something like this in the future?

- Document the code?
- Make it portable?

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

Final Exam

Final.

- 8:30am, Friday, January 19.
- A02 McDonnell Hall.

Reading period office hours.

- To be posted on Web.

Rules.

- No computational devices.
- Closed note, closed book.
- Exception: 8.5 x 11 page (both sides) in your own handwriting.

Tips for Preparing for the Final

Final is comprehensive.

Material since second midterm will be covered in greater depth.

- Theory.
  - Abstract machines: examples and applying theorems.
  - Computability: basic ideas and significance.
  - Analysis of algorithms (given code, predict how long it will take to solve problem)
  - P, NP, NP-complete, P = NP: basic ideas and definitions
- Systems.
  - basic ideas and definitions
  - understand examples

Be sure you understand questions you got wrong on previous exams.

- You just might get a similar problem…