6.00 Introduction to Computer Science and Programming

- Goal:
  - Become skillful at making a computer do what **you** want it to do
  - Learn computational modes of thinking
  - Master the art of computational problem solving

What does a computer do?

- Fundamentally a computer:
  - Performs calculations
  - Remembers the results
- What calculations?
  - Built in primitives
  - Creating our own methods of calculating

Is that all it does?

- A billion calculations per second

- 100s of gigabytes of storage

Are simple calculations enough?

- Searching the World Wide Web
- Playing chess
- Good algorithm design also needed to accomplish a task!

… so are there limits?

- Despite its speed and storage, a computer does have limitations
  - Some problems still too complex
    - Accurate weather prediction at a local scale
    - Cracking encryption schemes
  - Some problems are fundamentally impossible to compute
    - Predicting whether a piece of code will always halt with an answer for any input

Computational problem solving

- What is computation?
  - What is knowledge?
  - Declarative knowledge
    - Statements of fact
  - Imperative knowledge
    - "how to" methods or recipes

Declarative knowledge

- "The square root of a number `x` is a number `y` such that `y*y = x`"
- Can you use this to find the square root of a particular instance of `x`?

Imperative knowledge

- Here is a "recipe" for deducing a square root of a number `x` – attributed to Heron of Alexandria in the first century AD
  - Start with a guess, called `g`
  - If `g*g` is close enough to `x`, stop and say that `g` is the answer
  - Otherwise make a new guess, by averaging `g` and `x/g`
  - Using this new guess, repeat the process until we get close enough

An example

- Find the square root of 25

g	g*g	x/g	½(g + x/g)

How do we capture a recipe in a mechanical process?

- Build a machine to compute square roots
  - **Fixed Program Computers**
    - Calculator
    - Atanasoff and Berry's (1941) computer for systems of linear equations
    - Alan Turing's (1940's) bombe – decode Enigma codes
- Use a machine that stores and manipulates instructions
  - **Stored Program Computer**

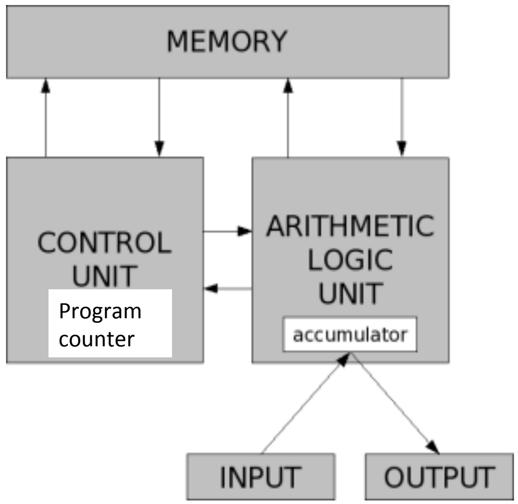
Algorithms are recipes

1. Put custard mixture over heat
2. Stir
3. Dip spoon in custard
4. Remove spoon and run finger across back of spoon
5. If clear path is left, remove custard from heat and let cool
6. Otherwise repeat from step 2

Stored program computer

- Sequence of instructions (program) stored inside computer
  - Built from predefined set of primitive instructions
    - Arithmetic and logic
    - Simple tests
    - Moving data
- Special program (interpreter) executes each instruction in order
  - Use tests to change flow of control through sequence, to stop when done

A basic machine architecture



Creating "recipes"

- Each programming language provides a set of primitive operations
- Each programming language provides mechanisms for combining primitives to form more complex, but legal, expressions
- Each programming language provides mechanisms for deducing meanings or values associated with computations or expressions

What are the basic primitives?

- Turing showed that using six primitives, can compute anything
  - **Turing complete**
- Fortunately, modern programming languages have a more convenient set of primitives
- Also have ways to abstract methods to create new "primitives"
- But anything computable in one language is computable in any other programming language

Aspects of languages

- Primitive constructs
  - Programming language – numbers, strings, simple operators
  - English – words
- Syntax – which strings of characters and symbols are well-formed
  - Programming language – we'll get to specifics shortly, but for example `3.2 + 3.2` is a valid Python expression
  - English – "cat dog boy" is not syntactically valid, as not in form of acceptable sentence

Aspects of languages

- Static semantics – which syntactically valid strings have a meaning
  - English – "I are big" has form <noun> <intransitive verb> <noun>, so syntactically valid, but is not valid English because "I" is singular, "are" is plural
  - Programming language – for example, <literal> <operator> <literal> is a valid syntactic form, but `2.3/'abc'` is a static semantic error

Where can things go wrong?

- Syntactic errors
  - Common but easily caught by computer
- Static semantic errors
  - Some languages check carefully before running, others check while interpreting the program
  - If not caught, behavior of program unpredictable
- Programs don't have semantic errors, but meaning may not be what was intended
  - Crashes (stops running)
  - Runs forever
  - Produces an answer, but not programmer's intent

Aspects of languages

- Semantics – what is the meaning associated with a syntactically correct string of symbols with no static semantic errors
  - English – can be ambiguous
    - "I cannot praise this student too highly"
  - Programming languages – always has exactly one meaning
    - But meaning (or value) may not be what programmer intended

Our goal

- Learn the syntax and semantics of a programming language
- Learn how to use those elements to translate "recipes" for solving a problem into a form that the computer can use to do the work for us
- Computational modes of thought enable us to use a suite of methods to solve problems