Course Logistics & Introduction

Theory of Computation

CISC 603, Spring 2020, Daqing Yun

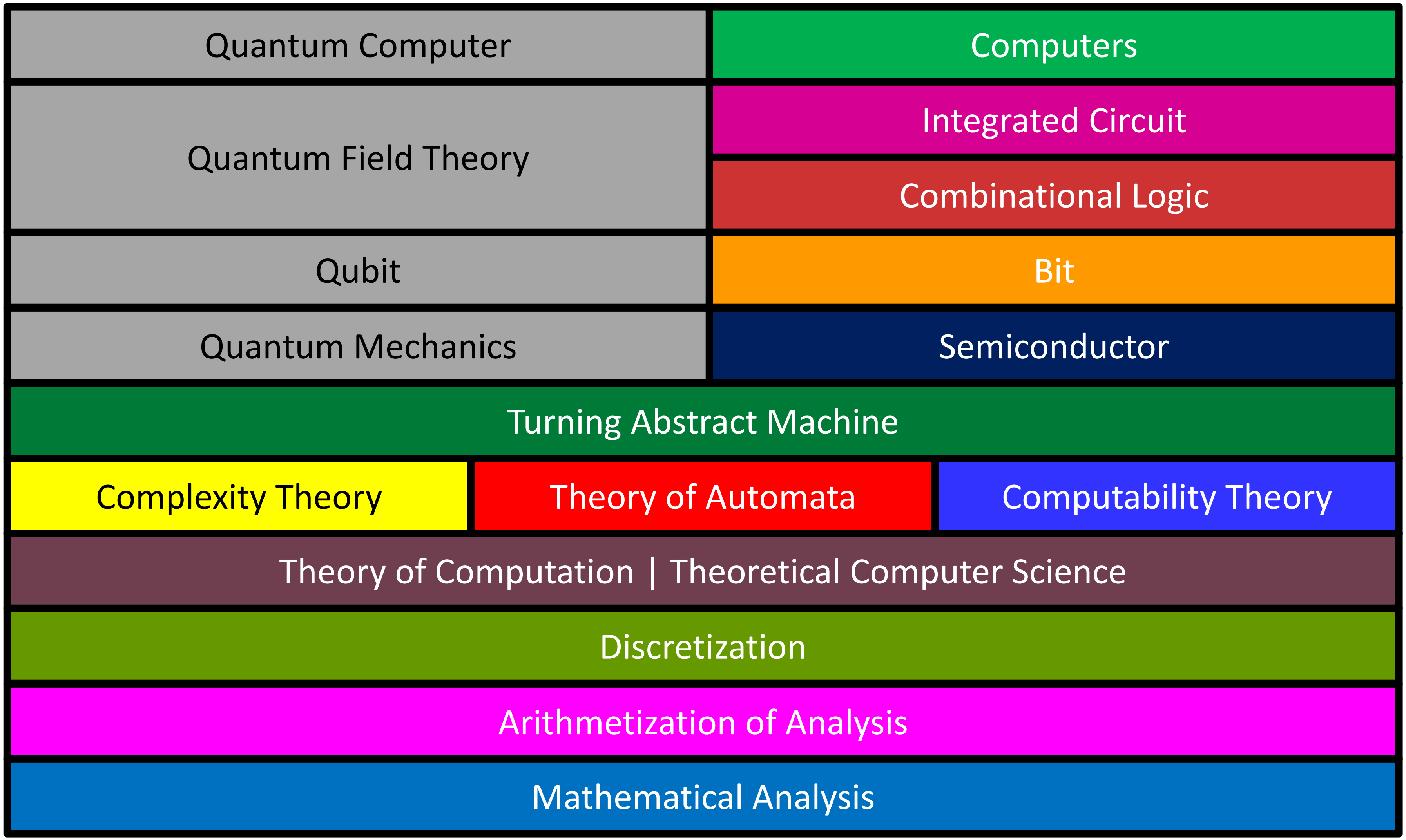
Who, Where, and When

- Daqing Yun, CISC program
- Office: 1227, HU main building
- Email: dyun@harrisburgu.edu
- Online meetings
  - Via AdobeConnect AdobeConnect
  - Thursdays, 6:00 – 8:00 pm EST
- See Canvas page for more details

The 1st In-Class Attendance Check

- Name
- Program (CSMS, ISEM, ANLY, etc.)
- Year
- Why do you take this course?
- Have you ever heard of
  - Finite automata
  - Regular expressions
  - Pushdown automata
  - Turning machines
  - Algorithms
  - Complexity
  - P, NP, NP-complete
  - …

Big Picture



Quantum Computer
Computers
Integrated Circuit
Quantum Field Theory
Combinational Logic
Qubit
Bit
Quantum Mechanics
Semiconductor
Turning Abstract Machine
Complexity Theory
Theory of Automata
Computability Theory
Theory of Computation | Theoretical Computer Science
Discretization
Arithmetization of Analysis
Mathematical Analysis

Problem Solving?

- What does it mean, really?
- What to do when we have a problem that looks easy?

If you can tell it is easy, then

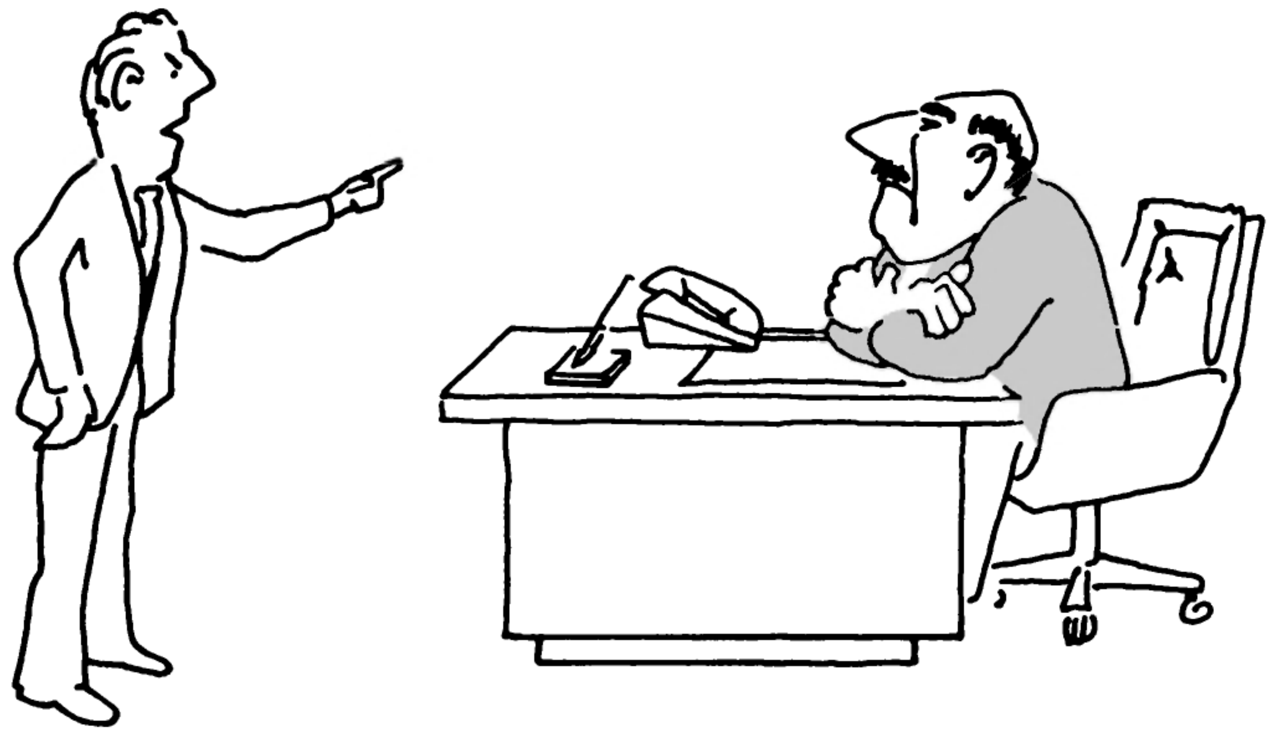
Problem Solving?

- What to do when we have a problem that looks hard?

"I can't find an efficient algorithm, I guess I'm just too dumb."

Problem Solving?

- What to do when we have a problem that looks hard?



"I can't find an efficient algorithm, because no such algorithm is possible."

Problem Solving?

- What to do when we have a problem that looks hard?

“I can’t find an efficient algorithm, but neither can these famous people.”

Sounds good, but …

- What is an “easy/hard” problem? – Define it!
- What is an “efficient” algorithm? – Define it!
- We cannot just simply say
  - “There is no such efficient algorithm” – prove it!
  - “These smart people cannot solve it either” – prove it!
- What is the best we can do?
  - Give up ☹ ? … or …
  - Design approximation algorithms?
    - What are these? – We know how it works for the worst case
    - What if the problem is not approximable? – prove it!
    - and then what to do? – heuristics

What will we learn in this course?

- What are the mathematical properties of computer hardware and software?
- What is computation, and what is an algorithm?
- Can we give rigorous mathematical definitions of these notions?
- What are the limitations of computers?
- Can everything be computed?
- Central question: what are the fundamental capabilities and limitations of computers?

Theory of Computation

- The question was asked by mathematicians in 1930’s when they were trying to understand the meaning of “computation”
- Whether all mathematical problems can be solved in a systematic way?
- Led to the computers as we know and use today
- Three areas:
  - Complexity Theory
  - Computability Theory
  - Automata Theory

Complexity Theory

- What makes some problems computationally hard and other problems easy?
  - Informally, a problem is called “easy” if it is “efficiently” solvable
  - Examples:
    - Sorting a sequence of, say, 1,000,000 numbers
    - Searching for a name in a telephone directory
    - Computing the shortest route to drive from HBG to where you live now

Example

Problem: Maximal Continuous Sub-array Sum Problem

Given: an integer array $A = a_1,\ a_2,\ \ldots,\ a_n$.
Output: the sub-array of A starts from index i and ends at index j that has the maximal sum.

-2	**-3**	**4**	**-1**	**-2**	**1**	**5**	**-3**
0	1	2	3	4	5	6	7

$O(n^3)$

Algorithm 1 BruteForceMaxSum(A)

```
maxSum = A[1]
for i = 1 to n do
    for j = i to n do
        sum = 0
        for k = i to j do
            sum+ = A[k]
        if maxSum < sum then
            maxSum = sum
return maxSum
```

Example

Problem: Maximal Continuous Sub-array Sum Problem

Given: an integer array $A = a_1,\ a_2,\ \ldots,\ a_n$.
Output: the sub-array of A starts from index i and ends at index j that has the maximal sum.

-2	-3	4	-1	-2	1	5	-3
0	1	2	3	4	5	6	7

$O(n^2)$

Algorithm 2 BruteForceMaxSumFaster(A)

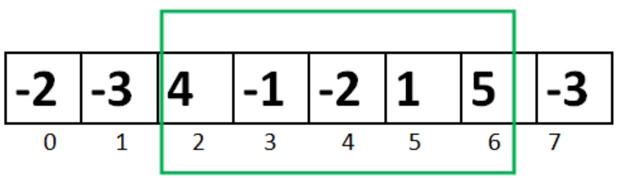
$maxSum = A[1]$
for $i = 1$ **to** n **do**
    $sum = 0$
    for $j = i$ **to** n **do**
        $sum += A[j]$
        if $maxSum < sum$ **then**
            $maxSum = sum$
return $maxSum$

Problem: Maximal Continuo

Given: an integer array $A =$

Output: the sub-array of A

the maximal sum.



$O(n\log_2 n)$

Algorithm 3 DivideConquerMaxSum(A)

procedure MaxSum(A, L, R)
    if $L < R$ **then**
        $m = \lfloor \frac{L+R}{2} \rfloor$
    $\mathcal{P}_1 = \text{MaxSum}(A, L, m-1)$
    $\mathcal{P}_2 = \text{MaxSum}(A, m, R)$
    $\mathcal{P}_3 = S = A[m-1]$
    for $i = m-2$ **to** L **do**
        $S = S + A[i]$
        if $S > \mathcal{P}_3$ **then**
            $\mathcal{P}_3 = S$
    $\mathcal{P}_4 = S = A[m]$
    for $i = m+1$ **to** R **do**
        $S = S + A[i]$
        if $S > \mathcal{P}_4$ **then**
            $\mathcal{P}_4 = s$
    return $\max\{\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3 + \mathcal{P}_4\}$
procedure DivideConquerMaxSum(A)
    $L = 1$
    $R = n$
    return MaxSum(A, L, R)

Example

Problem: Maximal Continuous Sub-array Sum Problem

Given: an integer array $A = a_1,\ a_2,\ \ldots,\ a_n$.
Output: the sub-array of A starts from index i and ends at index j that has the maximal sum.

-2	**-3**	**4**	**-1**	**-2**	**1**	**5**	**-3**
0	1	2	3	4	5	6	7

O(*n*)

Algorithm 4 DPMaxSum(A)

$max = a_1$
$\mathcal{P}_1 = a_1$
for $i = 2$ **to** n **do**
  if $\mathcal{P}_{i-1} > 0$ **then**
    $\mathcal{P}_i = \mathcal{P}_{i-1} + a_i$
  else
    $\mathcal{P}_i = a_i$
  if $max < \mathcal{P}_i$ **then**
    $max = \mathcal{P}_i$
return max

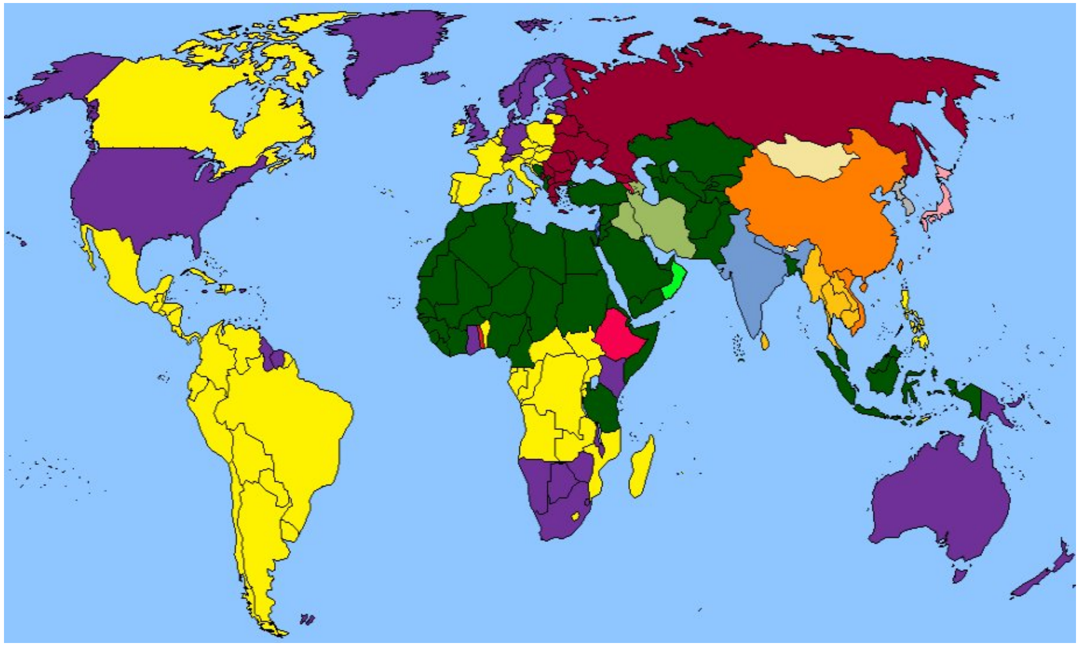
Complexity Theory

- What makes some problems computationally hard and other problems easy?
  - Informally, a problem is called “hard” if it cannot be solved efficiently, or if we do not know whether it can be solved efficiently
  - Examples:
    - Time table scheduling for all courses at HU
    - Factoring a 300-digit integer into its prime factors
    - Coloring maps using red, blue, green colors
    - Computing a layout for chips in VLSI

Example

Problem: Graph 3-Colorability Problem
Given: a graph $G = (V, E)$, an integer k=3 <= $|V|$
Question: can the vertices of G be colored using at most k colors such that adjacent vertices have different colors?

NP-complete



Complexity Theory

- Central question:
  - Classify problems according to their degree of “difficulty”
  - Give a rigorous proof that problems seem to be “hard” are really “hard”
  - Recall that what we can do when problem are hard (see Slide 9)

Computability Theory

- In 1930’s, Gödel, Church, and Turing discovered that some of the fundamental problems cannot be solved by a “computer”, which are only invented in 1940’s
- “Is an arbitrary math statement true or false?”
- Formal definitions are needed to tackle such a problem
  - Computer
  - Algorithm
  - Computation
- Theoretical models that were proposed in order to understand solvable and unsolvable problems led to the development of real computers

Computability Theory

- Central question:
  - Classify problems as being solvable and unsolvable
  - Can you write a program capable of predicting the behaviors of another program?

Automata Theory

- Deal with definitions and properties of different types of “computation models”
- Context-Free Grammars – Languages
  - Programming language definitions
- Finite Automata – Control
  - Text processing, lexical analysis
- Turing Machines – Hardware
  - A simple abstract model of a “real” computer, such as your PC at home

Automata Theory

- Central question:
  - Do these models have the same power, or can one model solve more problems than the other?
  - What kinds of languages can be recognized by NFAs, PDAs, TMs?

Introduction

- Formulating “theory of computation” threatens to be a huge project
- Narrow it down in its simplicity yet think systematically about what computers do
- Explaining foundations of theoretical CS in an engaging, practical way without assuming significant academic background
- “It receives some input, in the form of a string of characters; it performs some sort of “computation”; and it gives us some output”

Introduction

- Decision problems: questions can be answered either yes or no
  - “Is it a legal algebraic expression?”
  - The language accepted is the set of strings to which the computer answers yes
  - The language of legal algebraic expressions
- Computers play a role of a language acceptor
- Accepting a language is approximately the same as solving a decision problem
  - By receiving a string that represents an instance of the problem and answering either yes or no
- Many interesting computational problems can be formulated as decision problems

Introduction

- Finite automata – solve decision problems
  - Model: finite automaton
    - Proceeds by moving among a finite number of distinct states in response to input symbols, whenever it reaches an accepting state, answer “yes”
    - Lack of any auxiliary memory
  - Regular languages
    - Languages accepted by finite automaton
    - Regular expression or regular grammars
  - Pushdown automaton
    - More capable than finite automaton
    - Employs a stack
    - Generated by more general grammar context-free grammars

Introduction

- Pushdown automaton
  - More capable than finite automaton
  - Employs a stack
  - Generated by more general grammar context-free grammars
  - Can describe much of the syntax of high-level programming languages as well as legal algebraic expression and balanced strings of parentheses

Introduction

- Turning machines
  - The most general model of computation we will study
  - Carry out any algorithmic procedure in principle - as powerful as any computer
  - Accept recursively enumerable languages, generated by, e.g., unrestricted grammars
  - Not very “user-friendly”, leave something to be desired as an actual computer
  - Can be used as a yardstick for comparing the inherent complexity of one solvable problem to that of another
  - Problems that can be solved in a reasonable time and those that cannot could be distinguished by the number of steps a Turing machine needs to solve a problem – computational complexity

Introduction

- Turning machines
  - Simpler than any actual computer, because it is abstract
  - Enable our study of computation, without becoming bogged down by hardware details or memory restrictions
  - A TM is an implementation of an algorithm – studying one in detail is equivalent to studying an algorithm, and studying them in general is a way of studying the algorithmic method
  - Having a precise model makes it possible to identify certain types of computations that Turing machines cannot carry out
  - Accept recursively enumerable languages, not all languages
    - i.e., cannot solve every problem
    - Undecidable problems
    - A limitation of algorithmic method

Introduction

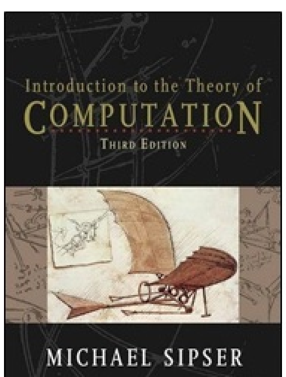
- An (unofficial) recipe of solving problems
  1. Problem formulation – specific about what to achieve
  2. Complexity analysis – how hard?
  3. Algorithm design – abstract solution
  4. Implementation – real-life solution
  5. Evaluation – performance

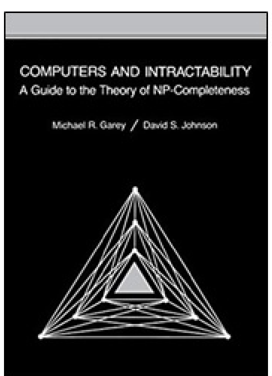
Topics

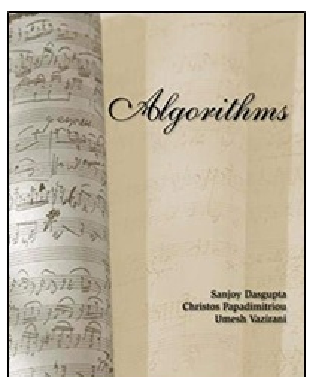
- Introduction – what we are doing now
- Finite automata and the languages they accept
- Regular expressions
- Nondeterminism
- Context-free languages
- Pushdown automata
- Turing machines
- Undecidable problems
- Computable functions
- Computational complexity

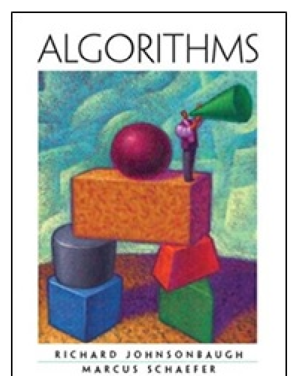
Textbooks and Reading Materials

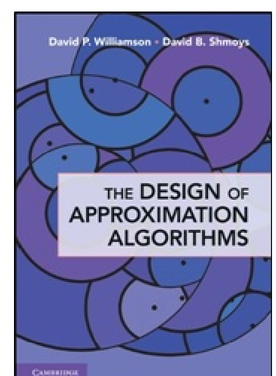
- **[Sipser]** M. Sipser. *Introduction to the Theory of Computation* (3rd Ed.), 2012, ISBN: 978-1133187790.
- **[GJ]** M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1979, ISBN: 0-7167-1044-7.
- **[DPV]** S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms*, 2008, ISBN: 0073523402.
- **[JS]** R. Johnsonbaugh and M. Schaefer. *Algorithms*, 2003, ISBN: 0023606924.
- **[WS]** D. Williamson and D. Shmoys. *The Design of Approximation Algorithms*, 2011, ISBN: 0521195276.
- **[Stuart]** T. Stuart. *Understanding Computation: From Simple Machines to Impossible Programs*, 2013, ISBN: 978-1449329273. http://computationbook.com

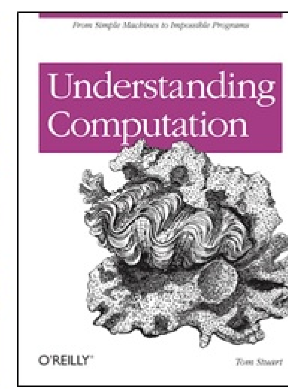












Course Project

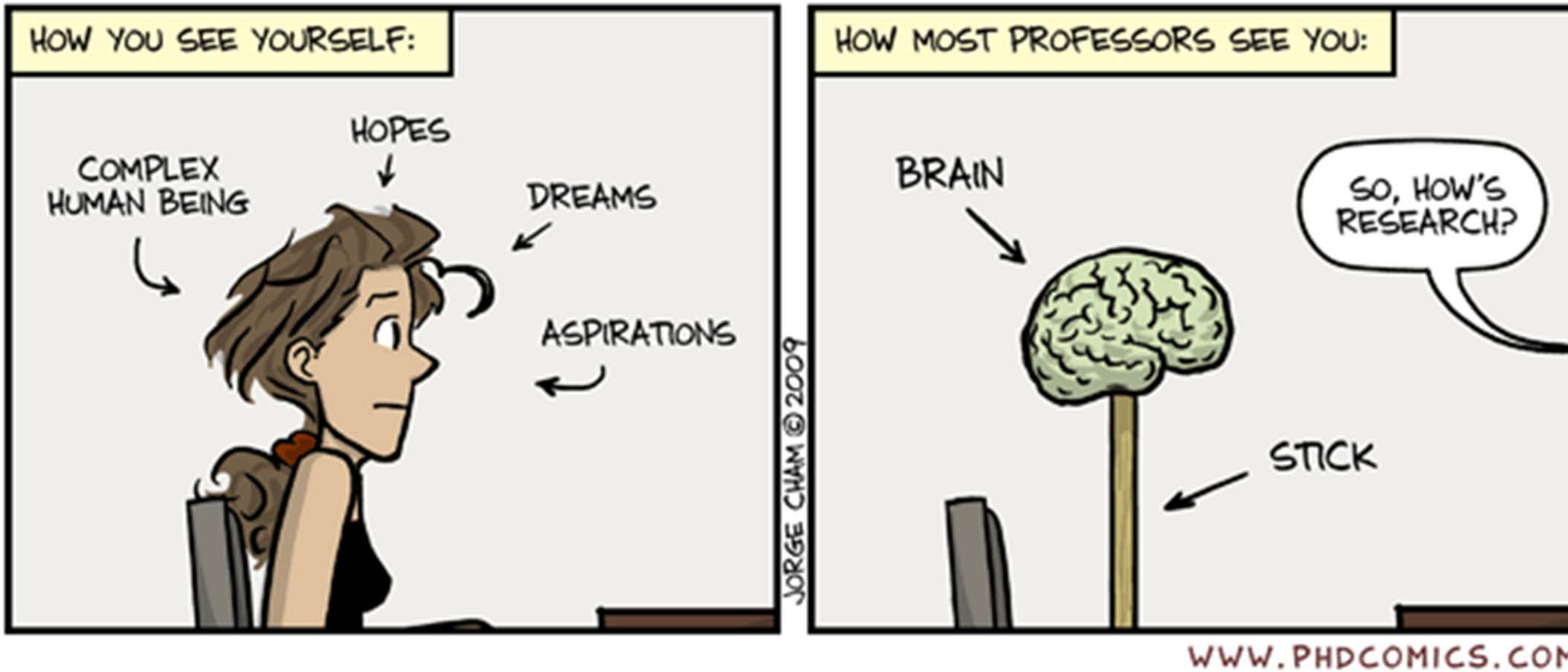
A Toy Compiler

- Programming challenge: write a program (using any language of your choice) that can "understand" and "execute" some commands you defined beforehand
- For example, see in-class demo

```
// this is a curve of sin function
origin is (200,200);
rot is 0;
scale is (10,4);
for T from 0 to 2*pi + pi/50 step pi/500 draw(T,-30*sin(T));
```

- Course project deliverable 1 asks you to set up your development environment

I know you have other things to do, but …



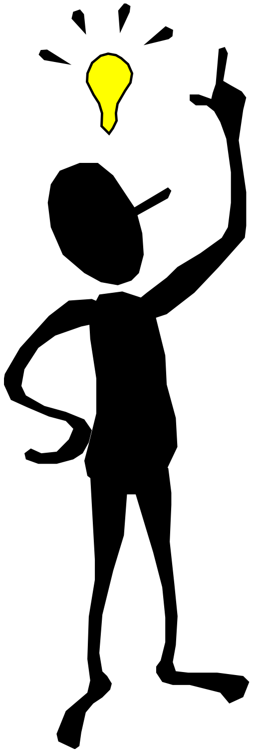
The first rule of CISC 603 is

- Don't plagiarize in CISC 603

- The second rule of CISC 603 is
  - Don't plagiarize in CISC 603

- Detection system
  - TurnItIn

- Penalty
  - 0 in this course, report to the University



Thanks ! ☺

Questions ?