Notes:

Note that often the first release of a document (for example, `ch0.pdf`) will only contain the first part of the chapter, so if you miss a physical handout, you'll need to look online to see if there are more pages that you don't have.

For this first class session, there was no preliminary reading.

Because this first session is introductory and exploratory, it has no *routine (or optional) projects.*

We will definitely finish this stuff in class today. To prepare for Session 2, you should read the brief document `bruteForce.pdf` (found at the course web site as soon as I finish it) and do the suggested Exercises. Note that these are not checked or graded in any way, but are intended to model for you how you should read technical stuff—basically doing some hands-on activity everywhere you can.

> Depending on your learning style, you may find it helpful to watch online videos instead of or in addition to looking at the "pre-reading" documents I provide. I'm going to focus on producing good written materials.

Whether you read my documents, or a text book, or watch videos, be sure to put in enough time getting ready for the next class session, or you might let down your group!

Introduction to Problems, Instances, and Algorithms

Our first work will be to begin thinking about problems and algorithms with some hands-on work with problems that most people would call "puzzles."

Usually the following material would be in the pre-reading, but since we didn't do any before hand (because there was no "before hand") here it is:

The idea of an *algorithm* is a fundamental concept that can't really be precisely defined (like "point" and "line" in geometry). From your previous CS courses (and life), you probably have a good sense of what it means. Levitin says that "an algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time."

This course will help you to be better at

learning a new algorithm someone else has designed,

designing new algorithms (something that you do, technically, every time you create a new program) using a number of design techniques,

analyzing the efficiency of a given algorithm,

implementing a given algorithm, and

dealing with the computational intractability of certain problems.

Some Informal Definitions

A *problem* is a collection of *instances* for which we hope to create an *algorithm* that will produce desired results when given an instance of the problem.

For example, the sorting problem is "given a finite sequence of items, each with a key, where the keys can be compared (in the sense of the `Comparable` interface in Java), rearrange the items so the keys are in some specified order (increasing or decreasing). An instance of this problem is "sort the numbers 3, 1, 5, 7, 11, 2, 4, 101, 26 into increasing order."

To *solve* the sorting problem means to create an algorithm that can produce the correct results—a sorted list—for any given input—an arbitrary list.

Exploring Some Problems

We are now going to explore some problems. For each of these, you will be asked to try to solve one or more given instances.

> These problems all fall into the category of "puzzles," because they typically only have a few—or even only one—instance, and they don't have much apparent purpose in real life (for most of the course we will focus on more practical problems, but puzzles can sometimes help us learn about algorithm design techniques).

As you try to solve them, try to be aware of whether you have an algorithm in mind, and if so, what it is. Technically, reaching the desired goal for an instance does not mean you have "solved" the problem—that requires having an algorithm that allows a mechanical process to reach the goal for any instance.

Be aware that some of these problems may be quite easy and others may be quite difficult. For some you may have clever insights leading to slick solutions, while for others you may be unable to solve them at all.

No matter what, please do not try to find the answers on the web during class today (you can look them up later, if you want)! The point is to explore the process of trying to figure out how to solve these problems.

If you are finding a puzzle interesting or think you are making progress toward solving it, go ahead, but don't hog any puzzle too long.

Tower of Hanoi

Start with a stack of n disks on one pole, with the disks in decreasing size as you go up in the stack (so there is an instance of this problem for each integral value of n from 1 up). Move all the disks from the pole they are on to one of the other two poles, using the rule that the only action you can take is to take one disk on the top of a stack and move it to the top of another stack (perhaps with 0 disks), never placing a disk on top of a smaller disk.

Fifteen Puzzle

Given some arrangement of the 15 tiles in the grid, slide one tile at a time until you restore the puzzle to the standard form:

```
 1     2     3     4
 5     6     7     8
 9    10    11    12
13    14    15     -
```

where − represents the empty space.

The Soma Cube

Given the seven pieces, arrange them into a 3 by 3 by 3 cube.

The L's Puzzle

The piece of foamboard provided gives three instances of this problem, namely 2 by 2, 4 by 4, and 8 by 8. An instance of the problem is one of the grids where your enemy picks any square to be left empty, and then you need to tile the grid with L blocks so as to cover all the other squares.

Pentominoes

This problem has four distinct instances.

Given a set of pieces known as pentominoes, arrange them to fill rectangles of size 3 by 20, 4 by 15, 5 by 12, and 6 by 10.

Instant Insanity

Given cubes numbered 1 through 4, with each face being red, blue, green, or white, arrange them in a row so that all four colors occur on each of the edges consisting of four squares (doesn't matter what colors show on the ends or where two cubes come together).
