CMPS 10
Introduction to Computer Science
Lecture Notes

Chapter 1: Introduction

What is Computer Science? Some possible misconceptions are:

- The study of computers
- Programming and programming languages
- Applications software like MS word, Adobe Photoshop, etc.

A widely accepted definition of computer science was given by Norman Gibbs and Allen Tucker: Computer Science is the study of Algorithms, especially their

(1) Mathematical Properties: Correctness, Efficiency, Complexity (time and space)
(2) Hardware Realizations: Logic Gates, Circuits, Architecture
(3) Software Realizations: Programming and Programming Methodologies
(4) Applications to Other Disciplines: Mathematics, Physics, Engineering, Business

Items (1), (2), and (3) constitute a rough outline of this course. We will not cover item (4), which is vast. So what is an algorithm? Informal definition: a step-by-step procedure which solves (all instances of) some specific problem. Some simple examples are: a cake recipe, instructions for balancing your checkbook, instructions for filling out your tax return

Example Consider the problem of adding two 3-digit numbers

```
  493        617
 +751       +945
 1244       1562
```

More generally we consider the problem of adding two m-digit numbers, where $m \geq 1$.
Input: $m \geq 1$ and two m-digit numbers: $a_{m-1}a_{m-2}\ldots a_2a_1a_0$ and $b_{m-1}b_{m-2}\ldots b_2b_1b_0$
Output: Their sum: $c_mc_{m-1}c_{m-2}\ldots c_2c_1c_0$

```
1) carry = 0
2) $i = 0$
3) while $i < m$ do 4-10
4)     $c_i = a_i + b_i +$ carry
5)     if $c_i \geq 10$ do 6-7
6)         $c_i = c_i - 10$
7)         carry = 1
8)     else do 9
9)         carry = 0
10)    $i = i + 1$
11) $c_m =$ carry
12) print $c_mc_{m-1}c_{m-2}\ldots c_2c_1c_0$
13) stop
```

Trace execution of this algorithm on: $m = 3$, $a_2 = 6$, $a_1 = 1$, $a_0 = 7$, $b_2 = 9$, $b_1 = 4$, $b_0 = 5$.

Note that the above operations can be classified into three types:

(1) Sequential Perform a single task, then move to the next operation in the list.
(2) Conditional (Branching) Select the next operation based value of a logical expression.
(3) Iterative (Loop) Repeat some block of instructions until some condition is met.

Why should we specify an algorithm in this way? Answer: If we can specify an algorithm in this level of detail to solve some problem, then we can automate the solution to that problem.

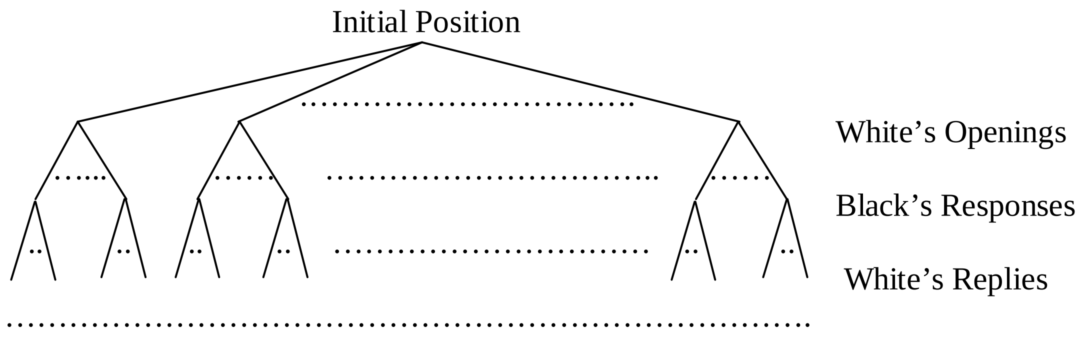
The entity (machine, robot, person, or persons) which executes the instructions in an algorithm is called the Computing Agent.

It may seem as if any problem which has a solution can be solved by some algorithm. In fact it has been shown that there are well defined problems, which have definite solutions, but for which no algorithmic means of reaching that solution can exist. (See Chapter 11, and references to Alan Turing, Kurt Gödel).

There are also problems for which we have algorithmic solutions, but these algorithms are so inefficient as to be unusable.

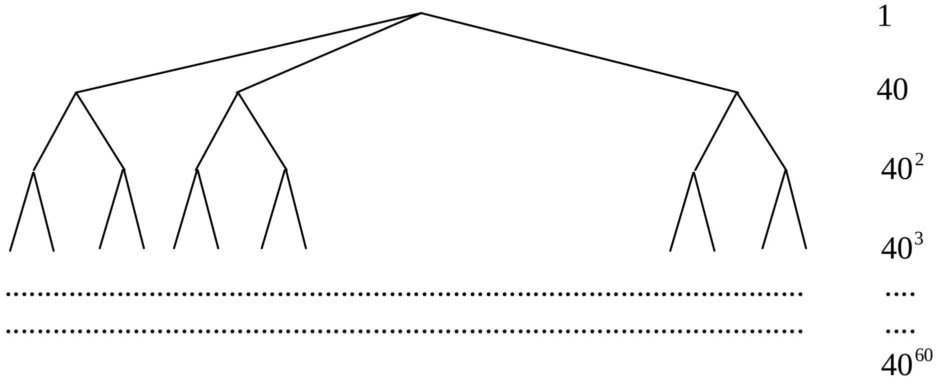
Example Brute force chess analysis

Consider an algorithm for playing the game of chess which operates as follows. For each of White's possible opening moves it considers every possible response by Black, and for each response by Black, it considers each of White's possible replies. The algorithm continues in this manner constructing the entire tree of possible game histories.



This tree is *very* large, we'll discuss how large shortly. Each descending path in this tree must end in a board position which is either a win for White, a win for Black, or a Draw (this follows from the rules of chess.) Once the tree is fully constructed, the algorithm can play what is essentially a perfect strategy for chess. If the algorithm is playing the role of White say, it simply avoids any moves which could lead to either a win for Black or a Draw. We call this algorithm *brute force chess analysis*. It would be a considerable challenge to write it down in pseudo-code (or any other formal language), but it can be done.

In order to appreciate the complexity of brute force chess analysis, let us estimate the size of the above tree. It is known that on average there are 40 legal moves from any board position, and on average a game of chess takes about 30 moves (i.e. 30 moves for White and 30 moves for Black, so 60 plys.) Therefore there are approximately 40 board positions at depth 1 in the tree, 40^2 positions at depth 2, and in general 40^k board positions at depth *k*. Since the average game has 60 plys, the algorithm must check approximately $40^{60} \approx 10^{96}$ ending positions to determine whether they are win for White, win for Black, or Draw.



Suppose the algorithm can evaluate 1 quintillion = 10^{18} board positions per second (a ridiculously high number.) The run time of the algorithm would then be

$$\text{Time} = \frac{10^{96}\text{ positions}}{10^{18}\text{ positions/second}} = 10^{78}\text{ seconds} \approx 10^{70}\text{ years}$$

The present age of the universe is estimated by cosmologists to be about 10 billion $= 10^{10}$ years. Thus the brute force chess analysis algorithm is utterly impractical.

Observe however that chess playing software is readily available. Obviously the underlying algorithm used by these programs is not brute force chess analysis. Indeed the problem with brute force chess analysis is that it seeks to find the perfect strategy. (Such a program would be no fun to play against anyway.) Instead chess programs use algorithms which are much closer to the way people play chess, i.e. they use a collection of heuristic principles which work most of the time, but which may fail in special circumstances.

This and other problems (see the "traveling salesman problem" in homework) serve to illustrate the limits of algorithmic problem solving. There are also problems which *may have* an algorithmic solution, but for which none is known. These tend to be problems which involve some notion of intelligence. The branch of Computer Science which deals with these problems is called Machine Learning or Artificial Intelligence.

<u>Formal Definition of Algorithm</u>
An algorithm is a *well ordered* collection of *unambiguous* and *effectively computable* operations that, when executed, *produce a result,* and *halt in finite time.*

Let us examine this definition in more detail. To say that a collection of instructions is *well ordered* simply means that it is clearly specified which operation to perform first, and when any operation is completed, which operation to perform next.

Example An algorithm which is not well ordered.

1) do something
2) do something
3) do something
4) repeat

Since it is not clear which steps to repeat, the above algorithm is not well ordered. Is *m*-digit addition well ordered?

An *unambiguous* operation is one that can be carried out directly by the computing agent with no outside assistance or further simplification. Such an operation is also called a *primitive operation* for the computing agent. Note that what is primitive for one computing agent may not be primitive for another. Our *m*-digit addition assumes that adding three single-digit numbers is a primitive operation. If 1-digit addition is *not* primitive, how can we specify an algorithm for it? Suppose our computing agent has access to a table with the answers to all 1-digit addition problems with three terms, and suppose looking up answers in this table is a primitive operation for our computing agent. Then line (4) of *m*-digit addition can be replaced by this lookup operation. If looking up answers in a table were not primitive we would have to specify yet more detail.

The level of detail/abstraction necessary to specify an algorithm depends on the capabilities of the computing agent. Is adding two 10-digit numbers a primitive operation for your pocket calculator?

An operation is *effectively computable* if it can be accomplished via some computational process. Essentially this means that it is doable by some computing agent.

Example $a = \sqrt{b}$

Observe that there are values of *b* for which no computing agent can perform this step (assuming the number assigned is to be a real number) i.e. take b negative.

Example list all prime numbers $p_{1,}\ p_2, p_3, \ldots\ldots$

No computing agent can finish performing this operation.

We say an algorithm *produces a result* rather than an answer because sometimes (i.e. for some inputs) there is no answer. In such a case the result would be an error message.

To say that an algorithm *halts in finite time* means that only finitely many operations will be executed.

Example The following is example is called an infinite loop.

1) do something
2) do something
3) go to (1)

This is why we always include the stop instruction in our algorithms. It should always be clear how and under what circumstances execution of an algorithm halts.

In summary we can say the Computer Science is the study of *Algorithmic Problem Solving*.