Models of Computation, Complexity Theory and Tetris

A Very Brief Introduction to Theoretical Computer Science (Maths in Disguise!)

December 4, 2015

1 What is Computer Science?

- Not glorified programming!
- Mathematical set of tools / ideas for understanding just about any system
- Start with simple systems, sets of rules that haven't been confirmed, but we assume to be true, then ask what sort of complex systems we can build (kind of the opposite approach to physics)

2 Models of Computation

2.1 Compass and Straightedge Constructions

- Computing circa 500 B.C.
- We define four rules:

  I Start with two points. Distance between them is unit length.

  II Can draw a line between any two points.

  III Can draw a circle given centre and a point on its circumference.

  IV Can draw a point at the intersection of any two lines.

- By applying these rules multiple times, we can construct all kinds of things - e.g. a unit square.
- The important idea here is *modularity* - once we work out we can bisect a line, we can just assume that it is always possible.
- More interestingly, we can create a polygon with $p \times q$ sides either where p is divisible by 2 or where p and q are coprime by simply expanding on the p and q-gons (the proof for this is not too hard)
- CS is also about investigating the limitations of our models - so for example the functions we can use here are both quadratic. Let's say we want to double the volume of a cube. I.e. given a side of length 1, we need to construct a line segment of $\sqrt[3]{2}$. It turns out (as you'd expect) that this is impossible (a more formal proof is possible with Galois theory). Another one you might like to try is proving that squaring the circle is impossible.

2.2 Logic and Circuits

- First we need to look at some notation: A implies B says "If A is true, then B must also be true" - note if A is false, then B could be true or false.
- Famous problem with 4 cards - let's say C,D,1,2. Which cards do we need to turn over in order to check the rule that if there's an C on one side there's a 1 on the other side? Only the A and the 2. (Interestingly if you phrase this as "who do we need to check given that you can only drink if you're over 18" it becomes much easier)

- Now look at a different problem. Digraphs can help you make sense of something like this, and spot any contradictions.
- If we add AND and OR, we have a complete set of logic operators - which brings us to our next model. Many of you will be familiar with logic circuits. We won't discuss these in detail but there are some fairly significant limitations - most notably that they can only be designed for a fixed size task.

2.3 Finite Automata and Regular Expressions

- A finite automaton is defined by a 5-tuple, made up of a finite alphabet (set of symbols), the finite set of all states, the initial state, a transition function that returns a new state given a symbol and current state, and a set of accepting states.
- We can use a finite state diagram to help show our transition function. Start with an alphabet of {0,1}. Let's create a finite automaton that will accept a word only if there are an odd number of 1's.
- Now let's try and create an automaton that accepts all palindromes. This is harder than we expected! Remember we can only have a finite number of states.
- Looking into this further requires the pigeonhole principle (if we have $N+1$ letters nd N pigeonholes, at least one pigeonhole must have two or more letters) - although it seems simple, it is non-trivial to prove through the systems of logic we discussed earlier - in fact, there is a famous theorem that states any proof of the pigeonhole principle based on logical statements requires a number of steps that increases exponentially with N!
- Assume that we can create a DFA to accept palindromes. Let's say our DFA has some number k states. Now let us take the string $O^k1^k0^k$. This must be accepted. But we have k transitions just for the first set of k 0's: i.e. starting at q_0, we visit $q_1, q_2, ..., q_k$. At least one of these states must be repeated since we require $k+1$ states and only have k to choose from. Somewhere, we require a loop of some length i. However, clearly we are now forced to accept something of the form $a^{k+i}b^ka^k$. And we get our contradiction.
- We call any expression recognised by a finite automaton a regular expression. (i.e. we have proven the language of palindromes is not regular) - here we have informally applied the "pumping lemma".

2.4 Turing Machines

- So to improve our model, let's allow the tape to move backwards and forwards. Let's also assume we can write to the tape or halt at any time. We now have a Turing machine. We now have pretty much unlimited memory and the capability of performing as much auxiliary computation as we need (not tied to our input).
- So at any given point, our turing machine has two questions - what value should we write down (if any), and should we move the tape left, right, or not at all?
- Again, we can formally represent this model using a set of $\{q, s, q', s', d\}$, where q is the current state, s is the current symbol, q' is the state the Turing machine should move to next, s' is the value to replace s and d is the direction in which to move the tape.
- In 1936, Turing's paper "On computable numbers" (in some sense the founding document of computer science) - also proved that we could build a turing machine that acts as an interpreter for another Turing machine - "existence of the software industry lemma"! Some competition to find the least number of states and symbols required - somewhere around 2 states and 3 symbols is the best so far.
- It's not too hard to prove a Turing machine can recognise a palindrome - just move between beginning and end of input. In fact, Turing machine is the most sophisticated model we have. Church-Turing thesis claims that anything computable can be computed by a Turing machine.

- Still, there are problems we can't solve - for example the halting problem - can we work out if a program is going to halt or not? The benefits of such a program are quite significant. E.g. we could prove or disprove Goldbach's Conjecture.
- Sadly, it has been proven this is not possible. Let H be a Turing machine that solves the halting problem, and M be a program with some input w (we don't know whether it will halt or not). H accepts if $M(w)$ halts and rejects if $M(w)$ runs forever. Now we create a Turing machine, H', that uses H to determine whether a program halts - if it does, H' enters a continuous loop and otherwise H' will halt. Now if we take H' and run it with the program H' with input H', we get a contradiction.
- We can also have a nondeterministic Turing machine - this will always take a path, where possible, to produce a "yes" answer (magic!). Essentially we have a Turing machine that makes very lucky guesses. Although seemingly impossible (can you engineer luck?), it proves a useful model to have as we will see later.
- I have covered just two levels of the Chomsky Hierarchy. In fact, there are four major models of computation, each of which is able to recognise a certain set of languages. In order from least computational power to most we have: finite automata (regular languages), pushdown automata (context-free languages), linear bounded automata (context-sensitive languages) and Turing machines (universal or recursively-enumerable languages). Each is a subset of the group that follows it.

3 Complexity Theory

3.1 Big-O Notation

- Big-O Notation gives an upper bound to the growth of a function as n increases - ignoring any coefficients or constants etc. Essentially the same as when you're taking the limit of a function to infinity, you can disregard any "small" terms, we do the same here.
- We use it to determine the order of the worst case (or sometimes average case) runtime of an algorithm in terms of the number of basic steps required and with respect to the size of the input, denoted n.
- We also have big-omega, big-theta, and little-o notation which mean similar but slightly different things. I won't discuss these in further depth.
- Common orders include:

  $O(k)$(constant) - e.g determining if binary number is odd

  $O(\log n)$ (logarithmic) - e.g. binary search

  $O(n)$ (linear) - e.g. finding an item in an unsorted list

  $O(n \log n)$ (linearithmic or quasilinear) - e.g. quick sort

  $O(n^k)$ (polynomial) - e.g. bubble and shuttle sort

  $O(k^n)$ (exponential) - e.g. solving TSP (but brute-force method is factorial!)

3.2 Complexity Classes

- Complexity classes are a method of grouping problems based on how "difficult" they are to solve. Let's start with P, EXP, and R. P is the set of all problems solvable in polynomial time by a deterministic Turing machine. Likewise EXP is the set of all problems solvable in exponential time by a deterministic Turing machine. As you'd expect, EXP includes significantly more difficult problems than in P, as well as all problems in P. R is the set of all problems solvable in finite time (problems with computable solutions).

- We concluded earlier that the halting problem is not in R. It actually turns out that most problems are not in R. This is provable. Let's consider a finite program as a finite string of binary characters (a natural number). Now let's consider a problem as a function from an input to a boolean value. This is equivalent to an infinite string of binary characters (a real number). The first infinity is smaller (countable) than the second (which is uncountable - as shown using Cantor's diagonal argument) - so disappointingly, we come to the conclusion that most problems are not in R!
- We discussed earlier non-determinism. Now we can define our final complexity class - NP - which is the set of problems solvable by a non-deterministic Turing machine in polynomial time. This is equivalent to the set of problems verifiable by a deterministic Turing machine in polynomial time. It easy to prove P-verifiable problems are NP-solvable, but slightly harder to prove in the other direction, so I will leave this as something for you to think about!
- Some seemingly obvious relationships are notoriously difficult to prove. P = NP? being an example. It would be nice to be able to solve every problem with a checkable answer in polynomial time - but it seems to most (sane) people that solving a problem is really much harder than checking the answer, for at least some problems (e.g. factorization). Likewise, no-one has been able to prove a relationship between NP and EXP - although it is much easier to prove that P $\neq$ EXP.
- NP-Hard problems are problems that are at least as hard as the hardest problems in NP. NP-Complete problems are problems that are both in NP and NP-Hard. Tetris is NP-complete, as we will see.
- Likewise, EXP-Hard and EXP-Complete have similar meanings - an example of this kind of problem is Chess.
- An important concept in determining completeness is reducibility. Say we have a problem A that can be converted into problem B in polynomial time (e.g. finding the shortest path on a weighted graph and on a non-weighted graph) - we can then say that A must be as hard as B.
- SAT is (kind of by definition) an NP-complete problem (although Cook's Theorem provides a formal proof). All other NP-complete problems can be reduced to SAT in polynomial time. One such problem is Tetris, which was found in 2004 to be reducible to 3-PRT, a well-known NP-complete problem.
- We have only scratched the surface of the 500+ complexity classes in existence. They can relate to both space and time complexity.
- Let's finish with a class called BQP (bounded error quantum polynomial) - Shor's algorithm is in this class. Obviously it includes everything in P, but its relationship with NP is still unknown. Complexity theory is still a very active area of research!
- Hopefully this has helped to link mathematics and computer science - some examples are cryptography (a lot of modular arithmetic!), machine learning (essentially applied statistics), and computational geometry amongst many.