Computational Complexity

As a model of computation, the Turing machine has provided us with a way to prove the existence of unsolvable (uncomputable) problems. Not only does the Turing machine help us find the limits of computability, but it can also help us classify problems that are computable - that have an algorithm for their solution - by the amount of work required to carry out the algorithm.

Finding the amount of work required to carry out an algorithm sounds like analysis of algorithms. Real algorithms are analyzed and classified as $\Theta(\log n)$, $\Theta(n)$, $\Theta(n^2)$, or what have you. By the Church-Turing thesis, any algorithm can be expressed in Turing machine form. In this form, the amount of work is the number of Turing machine steps (one per clock pulse) required in the worst case before the Turing machine halts. (We assume here that we are considering only tasks that "have answers" so that the Turing machine halts on all appropriate input.)

Turing machine computations are quite inefficient. Therefore if algorithms A and A' both solve the same problem, but A is expressed as a description of a Turing machine and A' as pseudocode for instructions in a high-level programming language, then comparing the number of operations each algorithm performs is rather meaningless. Therefore we assume that all algorithms are expressed in Turing machine form so that we can readily compare the efficiency of different algorithms.

Rather than discuss whether a Turing machine algorithm is $\Theta(n)$ or $\Theta(n^2)$, let us simply note whether it is a polynomial-time algorithm. (Only quite trivial algorithms can be better than polynomial time, because it takes a Turing machine n steps just to examine its tape.) Problems for which no polynomial-time algorithms exist are called **intractable**. Such problems may be solvable, but only by inefficient algorithms.

Definition: P

P is the collection of all sets recognizable by deterministic Turing machines in polynomial time.

Consideration of set recognition in our definition of P is not as restrictive as it may seem. Because the Turing machine halts on all appropriate input, it actually *decides*, by halting in a final or nonfinal state, whether the initial string was or was not a member of the set. Many problems can be posed as set decision problems by suitably encoding the objects involved in the problem.

For example, consider the Hamiltonian path problem of whether a graph has a simple path that uses every node of the graph. We may define some encoding process to represent any graph as a string of symbols. Strings that are the representations of graphs become appropriate input, and we want to decide, given such a string, whether it belongs to the set of strings whose associated graphs have Hamiltonian paths. If we can build a Turing machine to make this decision in polynomial time, then the Hamiltonian path problem belongs to P.

The Hamiltonian path problem is solvable by the brute-force approach of tracing all possible paths one after another, but this is an n! solution (even worse than exponential) because of the number of paths. There is no known efficient (polynomial) algorithm to solve the Hamiltonian path problem, so we have no proof that the Hamiltonian path problem belongs to P. But there is also no proof that the Hamiltonian path problem does not belong to P. Might a clever, efficient algorithm someday be found? To see why this is unlikely, consider another class of sets.

Definition: NP

NP is the collection of all sets recognizable by nondeterministic Turing machines in polynomial time. (NP comes from *n*ondeterministic *p*olynomial time.)

While a set in P requires that a deterministic Turing machine be able to **decide** (in polynomial time) about whether some string on its tape does or does not belong to the set, a set in NP only requires that a nondeterministic Turing machine be able to **verify** (in polynomial time) that a particular input string is in the set. Given a graph that has a Hamiltonian path, for example, this fact can be confirmed in polynomial time by a nondeterministic Turing machine that fortuitously picks the correct path and then checks that it is correct, so the Hamiltonian path problem belongs to NP.

If a deterministic Turing machine can decide in polynomial time whether an arbitrary string belongs to a set, it can surely use the same process to verify a member of the set in polynomial time. Therefore $P \subseteq NP$. However, it is not known whether this inclusion is proper, that is, whether $P \subset NP$ so that there could be NP problems - including perhaps the Hamiltonian path problem - that are intractable.

The Hamiltonian path problem belongs to a third class of problems known as **NP-complete problems**.

Definition: NP-complete

A set L is **NP-complete** if it is in NP and every other set L_1 in NP is **polynomial-time reducible** to L. This means there is a polynomial-time TM that transforms any w_1 in the alphabet of L_1 to a word w in the alphabet of L in such a way that w_1 is in L_1 if and only if w is in L.

The first example of an NP-complete problem is the **satisfiability problem**. Given a boolean expression in conjunctive normal form (product of sums form), is there an assignment of truth values that will make the expression true? We can see it is NP - guess a truth value assignment and check whether it works.

For a set L that is NP-complete, if a polynomial-time decision algorithm were ever found for it, that is, if it were ever found to be in P, then every NP problem would be in P. Take any NP set, transform it to the NP-complete problem L (polynomial time), then solve the problem L (polynomial time). Total is polynomial time (composition of polynomials). Then indeed we would have P = NP.

A large number of problems from many different fields have been found to be NP-complete since this idea was formulated in 1971, and no polynomial-time decision algorithm has been found for any of them. Therefore it is now suspected that $P \subset NP$ and that all these problems are intractable, but to prove this remains a tantalizing goal in computer science research.