Reductions-1

Throughout, if we say "machine" we mean deterministic machine, unless we specifically say "non-deterministic." If L_1, L_2 are languages over the alphabets Σ_1 and Σ_2 , respectively, a reduction from L_1 to L_2 is function $R : \Sigma_1^* \to \Sigma_2^*$ such that $R(w) \in L_2$ if and only if $w \in L_1$. In our discussion of $\mathcal{NP}$ and $\mathcal{NP}$–completeness, all reductions are computable in polynomial time.

Instances. A reduction from a 0/1 problem L_1 to a 0/1 problem L_2 need only be defined on instances of L_1 , since we define $R(w) = \lambda$ if w is not an instance of L_1 Reductions found in the literature or on the internet are typically defined only on instances. For example, let BOOL be the language of all Boolean expressions, over an appropriate alphabet. An assignment of an expression $E \in$ BOOL is a mapping of the set of all variables which appear in E to the Boolean alphabet $\{0, 1\}$, where 0 means false and 1 means true. The assignment is *satisfying* if replacing each variable by its assigned truth value causes E to become true. If E has a satisfying assignment, we call it *satisfiable*, otherwise E is a contradiction. SAT$\subseteq$BOOL is the set of all satisfiable Boolean expressions, while an instance of the Boolean satisfiability problem is any Boolean expression. A Boolean expression is in *conjuctive normal form* (CNF) if it is the conjunction (and) of clauses, each of which is the disjunction (or) of terms, each of which is either a variable or the negation (not) of a variable. CNF$\subseteq$BOOL is the set of all Boolean expressions written in conjunctive normal form, while k-CNF $\subseteq$ CNF is the subset of CNF where each clause has k terms.

A language L is in the class $\mathcal{P}$–TIME (or simply $\mathcal{P}$) if there is some constant k and some machine which decides L in $O(n^k)$ time.

A problem P is $\mathcal{P}$–TIME if the language of all true instances of P is $\mathcal{P}$–TIME. For example, CNF and 3-CNF are both $\mathcal{P}$–TIME. The CYK algorithm demonstrates that every context-free language is $\mathcal{P}$–TIME. A language L is in the class $\mathcal{NP}$–TIME, usally called simply $\mathcal{NP}$, if it is recognized in $O(n^k)$ time by some non-deterministic machine, where n is the number of bits in the input. Every deterministic machine is also a non-deterministic machine, thus $\mathcal{P} \subseteq \mathcal{NP}$. The converse is an open question.

Verifier Definition of $\mathcal{NP}$

There is an equivalent definition of $\mathcal{NP}$ which is much easier to work with, the verifier definition, given below. Let $L \subseteq \Sigma^*$ be a language. Then L is $\mathcal{NP}$ if and only if there is an integer k and a program V (called the verifier) such that:

1. The input of V is an ordered pair of strings (w, c) and V executes in $O(n^k)$ time for some k, where n is the length of w, and the output of V is Boolean.

2. If $w \in L$ there is a string c, called a *certificate*, or *witness* for w, such that, with input (w, c), V returns 1.

3. If $w \notin L$, V returns 0 with input (w, c) for any string c.

NP–Completeness

We define a language L to be $\mathcal{NP}$–complete if the following conditions hold.

1. L is $\mathcal{NP}$,
2. . If L_2 is any $\mathcal{NP}$ language, there is a polynomial time reduction of L_2 to L .

Is $P = \mathcal{NP}$?

Theorem 1 *If there is any language which is both $\mathcal{P}$–*TIME *and $\mathcal{NP}$–complete, $\mathcal{P} = \mathcal{NP}$.*

Proof: We already have $\mathcal{P} \subseteq \mathcal{NP}$. Let L_1 be both $\mathcal{P}$–TIME and $\mathcal{NP}$–complete. We need to show that any $L_2 \in \mathcal{NP}$ is in $\mathcal{P}$. Since $L_1 \in \mathcal{P}$, there is a machine M_1 that decides L_1 in polynomial time. Since L_1 is also NP-complete, there is a polynomial time reduction of L_2 to L_1 computed by some machine M_2. Let the output of M_2 be the input of M_1 The combined machine decides L_2 in polynomial time. □

Theorem 2 below shows that every $\mathcal{NP}$ problem is decidable.

Theorem 2 *Any $\mathcal{NP}$ problem can be decided by a deterministic machine with polynomial memory in exponential time.*

Guide Strings. Recall the story of Ariadne, daughter of King Minos, rescuing Theseus from the Labyrinth, where he and eleven other youths and maidens were to be devoured by the Minotaur, who was half man and half bull. She brought him a sword, and he killed the Minotaur. Fortunately, she had tied one end of a string to the entrance and carried the other end to Theseus, and they were able to find their way out of the Labyrinth. That was the first guide string!

Proof: (Of Theorem 2) If L is recognized by a non-deterministic machine M in polynomial time, M can accept a string $w \in L$ deterministically in polynomial time by using a guide string, a sequence of instructions which tell which choice to make at each step of the computation. Given an arbitrary string $w \in \Sigma^*$, where Σ is the alphabet of of L, any such guide string must have polynomial length. We can assume each guide string is written only in binary. We need only try every possible guide string of that length. Each computation of M guided by a guide string takes polynomial time and polynomial memory, and the guide strings can be tried in canonical order by using only polynomial memory to keep track of the last one tried. If none work, $w \notin L$. The number of possible guide strings of length m is 2^m, hence exponential time is sufficent for this task. □

Consequences of $\mathcal{P} = \mathcal{NP}$

If $\mathcal{P} = \mathcal{NP}$, there is no one-way function, which implies that cryptographic systems which rely on functions believed to be one-way can be broken in polynomial time.