Chapter 2

Reductions, or *When is one problem no harder than another?*

2.1 Ordering Problems by Hardness

When is one problem harder than another? We want to say that problem B is harder than problem A when we have to use more resources (like time or space) to solve B then we need to use to solve A. Perhaps I have an $\Theta(n^3)$ algorithm for B and an $\Theta(n^2)$ algorithm for A. Am I justified in claiming that B is harder ha problem A? Let's be definite: NO, I'm not justified.
All I know is that one algorithm for B takes more time than some other algorithm for A. If I'm sensible, and try to look up algorithms for B, perhaps I'll find an $\Theta(n \log n)$ algorithm. Now I have an $\Theta(n \log n)$ algorithm for B and an $\Theta(n^2)$ algorithm for A. Should I change my mind and now claim that "A is harder than B"? In our everyday sense of "hard" this might be reasonable. That is, the hardness or easiness of a problem might depend on my state of knowledge. When I know more about a problem, I may be willing to say that it is easier than I thought. While such locutions might make sense in casual conversation, in theory we would like the hardness of a problem to be independent of my feelings or state of knowledge. Our theory should be Platonic in the sense that hardness exists in the eternal abstract nonmaterial mathematical universe. Our theory should not depend on our feelings or the limitations of our knowledge about problems. [1]

As we have said, we may not and usually do not know the BEST algorithm

[1] Various philosophers might argue that no such abstract universe exists and that real thought only exists in our real material bodies and brains. To them we reply that the assumption of the Platonic universe of mathematical objects has proven very effective in practice, and so even if the abstract universe does not exist, it may be practically useful to adopt this fiction in our reasoning.

for a problem. Even if we know the BEST algorithm, we may not know that it is best. To skirt around this difficulty of lack of knowledge, we can try to relate one problem to another. This is the general approach with the unfortunate (regrettable) name – REDUCTION. We will eventually talk about problem A being *reduced* to problem B.

The problem with this terminology is that we are trying to say that A may be easier than B, or as I prefer to say

A is no harder than B.

We can symbolize this relationship by the usual "less than" symbol

$$A \leq B.$$

Reading this as *A is no harder than B* has the potentially easier problem A and the potentially harder problem in the "right" places. Reading this as *A reduces to B* tends to cause the confusion that we usually reduce something harder to something easier and so it seems that A is the harder problem and B is the easier problem.

2.2 Partial Orders

We want to discuss an ordering on problems which is analogous to the less than ordering on numbers. But, we expect our problem ordering to have some different properties than our number ordering. Let us recall some facts and definitions about orderings (or orders). A *relation* on a set S is a set of ordered pairs of elements from S. [2] An *ordered pair* has a first element and a second element, and which is first and which is second matters. So, for example, (1,3) is an ordered pair of integers which is different from (3,1) because the first of these pairs has 1 as its first element and the second of these pairs has 3 as its first element. Two ordered pairs are the same if they both have the same first element and they also both have the same second element.

Some simple examples of relations are:

$$R1 = \{\,(1,2),\ (3,3),\ (1027,8)\,\}$$

$$R2 = \{\,(44,43),\ (42,41),\ (0,0),\ (41,42)\,\}$$

$$R3 = \{\,(1789,1776)\,\}$$

$$R4 = \{\,(APPLE, ORANGE),\ (SPINACH, STEAK)\,\}$$

$$R5 = \{\,(x,y) \mid y\ =\ x^2\ \}.$$

In these examples, we did not specify the set S because S is usually clear from the context in which a relation is defined. $R1$, $R2$, $R3$ and $R5$ presumably have

[2] Actually this is a *binary relation* since it only involves two elements at a time. We could have a more general view of relations which would involve k elements at a time. Since, we'll be using only binary relations, we'll use the term relation without the modifier "binary".

S's which contain numbers. I would guess that $S1$, $S2$, $S3$ and $S5$ contain the natural numbers, but they might contain many more numbers or many fewer numbers. As $R4$ points out, relations are not restricted to numbers, and an S could contain both numbers and non-numbers . Although in $R3$ the elements appear to be numbers, they could instead be dates, e.g. 1789 (the year of the French revolution) is related to 1776 (the year of the American revolution). $R1$, $R2$, $R3$ and $R4$ are finite relations because they contain only a finite number of ordered pairs. Such finite relations can often be specified by listing their ordered pairs. $R5$ may be a finite or infinite relation depending on $S5$. If $S5$ were the natural numbers, then $R5$, which is the relation between a number and its square, would be an infinite relation. If $S5 = \{0,1\}$, then $R5$ would be the finite relation $\{(0,0),\ (1,1)\}$. It's also possible that $S5$ could contain some elements for which the squaring relation is not defined, and such elements could not appear as the first elements of any pairs in $R5$. The style of $R5$, representing a relation by a property is often used even for finite relations because the number of pairs may be too many to conveniently write down.

We would like our relations to have certain extra properties. Often we want relations to be *reflexive*, that is, if

$$\forall x \in S \quad (x,x) \in R$$

then R is a **reflexive** relation. For orderings, the most important property is *transitivity*. A relation R is **transitive** if

$$\forall x,y,z \in S \quad (x,y) \in R \text{ and } (y,z) \in R \quad \text{implies} \quad (x,z) \in R.$$

A **partial order** is both **reflexive** and **transitive.** [3]

An additional property of many relations is symmetry. A relation, R, is **symmetric** if

$$\forall x,y \in S \quad (x,y) \in R \quad \text{implies} \quad (y,x) \in R.$$

For a symmetric relation we can forget about the pairs being ordered, and just say that a symmetric relation is a set of (unordered) pairs.

A relation which is **reflexive** , **symmetric**, and **transitive** is called an **equivalence relation.** For example, the usual equality relation on natural numbers is an equivalence relation because it consists only of the pairs (x,x) for every nartural number x. Obviously, this equality relation, $EQUAL$, is reflexive and symmetric. It is also transitive because

$$(x,x) \in EQUAL \text{ and } (x,x) \in EQUAL \quad \text{implies} \quad (x,x) \in EQUAL,$$

and there are no other pairs which have x as a second element. Many other relations are also equivalence relations. For example, $MOD2$ on natural numbers is the relation which says that $(x,y) \in MOD2$ iff x and y leave the same remainder (either 0 or 1) when divided by 2. Here, again reflexive and symmetric

[3] With only these two properties, some authors call the relation a pre-order and reserve the term partial order for a pre-order with one additional property.

are obvious. $MOD2$ is also transitive, because if x and y leave remainder r, and y and z leave remainder r', then $r = r'$ and so x and z both leave remainder r.

In attempting to make partial order similar in definition to equivalence relation, some authors add the anti-symmetric property to partial orders. A relation, R, is **anti-symmetric** if

$$\forall x, y \in S \quad (x,y) \in R \text{ and } (y,x) \in R \quad \text{implies} \quad x = y.$$

The usual less than relation on integers has this anti-symmetric property. That is,

$$x \leq y \text{ and } y \leq x \quad \text{implies} \quad x = y.$$

(Notice that we have changed the way we are writing relations. Instead of writing $(x, y) \in R$, we write $x \ R \ y$ where R is the name of the relation or a synbol representing the relation.)

What bothers me about this definition is that a relation can be *both* symmetric and anti-symmetric. For example, the usual equality relation is both symmetric and anti-symmetric. That is,

if $x = y$ then $y = x$ (symmetric),

but also, if $x = y$ and $y = x$ then $y = x$ (anti-symmetric).

Further, we run into some relations which I would like to call partial orders which are not anti-symmetric. For example, consider a directed graph in which we have $x \rightarrow y$ iff there is a directed edge from x to y. I would like to say $x \leq y$ if these is a chain of (zero or more) directed edges so that

$$x \rightarrow y_1 \rightarrow y_2 \ldots \rightarrow z.$$

While this $\leq$ is reflexive (using zero edges) and transitive by construction, it may not be anti-symmetric because there may be directed cycles in the graph. In the special case, when there are no directed cycles in the graph, we call the graph a **DAG**, a directed acyclic graph, and $\leq$ on a DAG is anti-symmetric. But, if there are directed cycles, say $x \rightarrow y$, $y \rightarrow z$, and $z \rightarrow x$, then $x \leq z$ and $z \leq x$ but $x \neq z$, and $\leq$ is not anti-symmetric.

There is a standard method to skirt this problem. We first define an equivalence relation on elements, then form equivalence classes of elements, and then put a partial order on these equivalence classes rather than on the original elements. Continuing with our digraph example, we define the equivalence relation $\equiv$ by $x \equiv y$ iff there is a directed cycle which contains both x and y. This relation is reflexive using zero length cycles. The relation is symmetric because its definition is symmetric in x and y. Finally, this relation is transitive because if there a directed cycle containing both x and y then there is a directed path from x to y, and similarly if there a directed cycle containing both y and z then there is a directed path from y to z and putting these two paths together gives a directed path from x to z. Turning this argument around, there is also a directed path from z to x and so there is a directed cycle which contains both x and z. (Notice that the paths and cycles we are using do not have to be simple – we allow vertices and/or edges to be used more than once.) The equivalence classes for $\equiv$ are sets of vertices. Two vertices, x and y, are in the same equivalence class exactly when there is a directed path from x to y and a directed path from y to x. (In graph theory, these equivalence classes are called the **strongly connected components** of the graph.) Now consider the digraph which has these equivalence classes as its vertices, and has a directed edge from equivalence class E_1 to equivalence class E_2 when there is some (original) vertex in E_1 which has a directed edge to some (original) vertex in E_2. If D is the original digraph and $\equiv$ is this equivalence relation, then we can use the notation $D/\equiv$ to refer to this new directed graph on equivalence classes. The point of this whole construction now becomes clear: $D/\equiv$ is a DAG and the $\leq$ on $D/\equiv$ is anti-symmetric. Specifically, if $E_i \to E_j$ and $E_j \to E_i$ then $E_i = E_j$. Or considering the original vertices within the E's, if $E_i \to E_j$ there is a vertex x in E_i which has a directed edge to some vertex y in E_j, and if $E_j \to E_i$ there is a vertex w in E_i which has a directed edge to some vertex z in E_i. But from the definition of $\equiv$, since y and w are both in E_j there is a directed path from y to w and thence to z and back to x, and so $x \equiv y$. This argument can be extended to show that for each $x_i \in E_i$ and $y_j \in E_j$, there is a directed path from x_i to y_j and a directed path from y_j to x_i, and so every element of E_i is equivalent to every element of E_j. Since these are equivalence classes, they are identical, and anti-symmetry is verified. While this construction always works, it gives partial orders defined on equivalence classes of elements rather than on the original elements themselves. To avoid this construction, I would prefer to redefine "anti-symmetric". First, I say that

$$\text{if } x \leq y \text{ and } y \leq x \quad \text{then} \quad x \equiv y.$$

(Here, I'm sneaking in the equivalence relation.)
Then, I say that $\leq$ is *anti-symmetric*

$$\text{if } x \leq y \text{ and } y \leq x \quad \text{implies} \quad x \equiv y.$$

So, by my definition the original relation on elements is automatically anti-symmetric if the relation is reflexive and transitive. Thus, a partial ordering need only be reflexive and transitive.

2.3 The General Idea of Reductions

At its core most modern science is reductive in the sense that it solves problems (or answers questions) by reducing them to problems or questions about smaller or simpler objects. In computer programming, in a similar way, we reduce problems to very simple problems by building complex systems out of a handful of simple primitives. The whole system works correctly if the primitives work correctly and if the reduction (our program) correctly reduces the problem to the primitives. In the overwhelming majority of cases, failures of systems are the result of programming errors rather than failure of the primitives to be correct. (The Pentium debacle of a decade ago is one of the few examples in which a primitive (a multiplier) was incorrect.) While the process of creating reductions is neither trivial nor well-understood, we still manage to get some programs to work correctly.

We are now going to take a step upward in the design process. Let us assume that we have correctly working programs for some problems. We might have designed these programs, or they might have been given to us by a wise guru, or we might have purchased them from a (we hope) trusted source, or we might have received these program as a divine gift, or (and this is usually the case when we are arguing complexity results) we might assume that these programs exist without actually having them or knowing how to construct them. We'd like to use these assumed correct programs to create programs for other problems.

The idea of reductions has two sources. One, which we have discussed in Chapter 1 is the classification of problems. There we hinted at the idea that the halting problem was, in some sense, the hardest problem in **RE**. The second source is traditional design of algorithms in which we solve one problem by solving another problem. For example, we can find the smallest element in an array by sorting the array and reporting the first element Here we can say that **Find-Smallest** is no harder than **Sort** and use the usual less than or equal to notation:

$$\textbf{Find-Smallest} \quad \leq \quad \textbf{Sort} \ .$$

Turing introduced the idea of an ordering on problems in the context of showing that a seemingly harder problem was no harder than a seemingly simpler problem. For example, Turing argued that **HALT** was no harder than **HALT-ON-ZERO**, but since **HALT-ON-ZERO** is a special case of **HALT**, and therefore simpler than **HALT**, he referred to his argument as a *reduction.* While later authors have used the ordering notation:

$$\textbf{HALT} \quad \leq \quad \textbf{HALT-ON-ZERO} \ ,$$

they have stuck with the word "reduction" and read this formula as

$$\textbf{HALT} \ \text{ is reducible to } \ \textbf{HALT-ON-ZERO} \ ,$$

or as

$$\textbf{HALT} \ \text{ reduces to } \ \textbf{HALT-ON-ZERO} \ .$$

I think that it is easier to read this as

$$\textbf{HALT} \ \text{ is no harder than } \ \textbf{HALT-ON-ZERO} \ ,$$

because this helps keep the harder problem on the right side of the $\leq$ sign, and when using "reduces" it is too easy to mistakenly put the easier problem on the right of the $\leq$ sign.

While it's obvious that

$$\textbf{Find-Smallest} \quad \leq \quad \textbf{Sort} \ ,$$

it takes a little effort to show that

$$\textbf{HALT} \ \leq \ \textbf{HALT-ON-ZERO} \ .$$

We begin by stating these two problems as **YES/NO** problems:

HALT
INPUT: A pair (M, t) where M is a program and t is an input for M.
QUESTION: Does the program M eventually halt when given the input t?

HALT-ON-ZERO
INPUT: A program M.
QUESTION: Does M halt when given input 0?

We want to take the input (M, t) for **HALT** and convert it into another program, say M_t, so that M_t halts on 0 exactly when M halts given t. We build M_t by starting with some code which looks at the input and if the input is 0, then this code overwrites the input with the input t. We add to this code the code for M, so that, M_t given 0 will overwrite 0 with t and then will behave like M given t. In particular, if M_t given 0 halts, then M given t will also halt. Conversely, if M_t given 0 does not halt, then M given t will not halt. Hence if we could answer the question: Does M_t halt when given 0 ?, we would also have the answer to the question: Does M eventually halt when given the input t?

As we said above, the word "reduction" seems to be appropriate here because **HALT-ON-ZERO** is a special case of **HALT**. That is, we could decide if M_t given 0 halts by correctly deciding whether $(M_t, 0)$ is a **YES** instance of **HALT.** So here in a very reasonable sense we have both

$$\textbf{HALT} \ \leq \ \textbf{HALT-ON-ZERO} \ ,$$

and

$$\textbf{HALT-ON-ZERO} \ \leq \ \textbf{HALT} \ .$$

In at least partial distinction from this, we saw that

$$\textbf{Find-Smallest} \ \leq \ \textbf{Sort} \ ,$$

but we would probably be willing to say that

$$\textbf{Sort} \ \not\leq \ \textbf{Find-Smallest}$$

because we can find the smallest element without sorting the array.

2.4 Some Examles

GENERAL IDEA of Reduction:

(a) $Sort \geq Find-Max$

(b) $Mult \geq DIV$

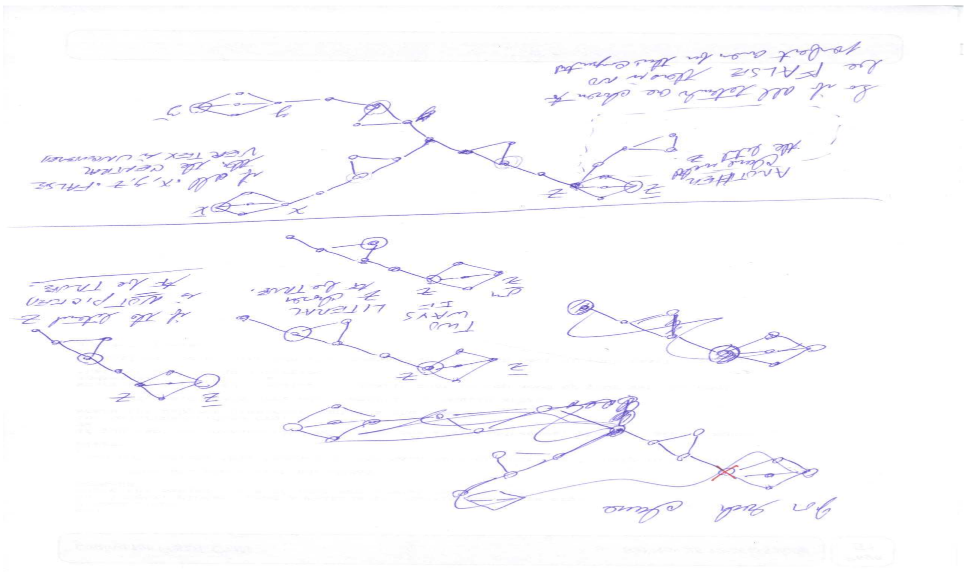
(c) $DIV \geq SQRT$

(d) Time preserving Reduction

(e) $Find-Max \geq Sort$ (Takes alittle more time.)

INCLUDE: The Hardest Context Free Lang
Reduction via Homo.
does NOT give diagram with REGULAR in the middle circle
Some Regular sets are NOT homomorphic
(E.G. Over {1} the strings of length = 0 (mod 2) are NOT homomorphic to the strings of length = 0 (mod 3))
Finite State Transformation restores PPicture.

INCLUDE examples like
One-error Correcting Codes



Regular Expression w/o Star

Reduce **SAT** to **3-SAT**
Trick: for a clause with four literals, e.g.

$$(z_1 \vee z_2 \vee z_3 \vee z_4)$$

introduce the new variable y and split the clause into two clauses:

$$(z_1 \vee z_{12} \vee y)\ (\bar{y} \vee z_{13} \vee z_4).$$

FILL IN THE REST OF THE ARGUMENT.

2.5 Turing Reductions

2.6 Many-one Reductions

Closure of RE. (Any set many-one reducible to an RE set is an RE set. In particular any set reducible to HALT is an RE set.)

2.7 Computationally Restricted Reductions

2.7.1 Primitive Recursive Reductions

Show that Prim Acc implies Prim Rej implies Prim Rec and vice-versa.

There is a RECURSIVE set which is NOT primitive recursive.

All non-trivial primitive recursive sets are mutually reducible.

Under RECURSIVE reduction all recursive sets are equivalent.

2.7.2 Polynomial Time Reductions

2.7.3 Finite State Reductions

2.7.4 Homomorphisms

2.7.5 Other Reductions

2.8 Complete Problems

Here we want to use the ideas of reduction to show that some classes of problems have *hardest* problems. The idea is that the *hardest* problems should be in the class and that all problems in the class are reducible to (*no harder than*) the hardest problem. We will call such problems **complete problems**. One caveat, it is traditional to say that problem B is C-complete to mean that B is hardest problem in the class C, but the notion of reduction being used is usually *implied* rather than stated.

2.8.1 HALT is RE-Complete.

2.8.2 LINEAR–HALT is $\mathcal{NP}$-Complete.

Here we want to show that there is an $\mathcal{NP}$-Complete problem. That is, we will give a problem B and show that it is the hardest problem in $\mathcal{NP}$. As we mentioned above, we have to decide which notion of ordering among problems we should use.

2.8.3 $\mathcal{NP}$-Hard Problems.

A problem $\mathcal{C}$ is called *$\mathcal{NP}$-hard* if an $\mathcal{NP}$-complete problem $\mathcal{B}$ is *polynomial time Turing reducible* to $\mathcal{C}$.

Often, there is some implication that $\mathcal{C}$ is is not *too* hard, meaning that $\mathcal{C}$ can be solved by a polynomial time non-deterministic algorithm, even though $\mathcal{C}$ may not be a **Yes/No**-problem. If this is the case, some authors call $\mathcal{C}$ an $\mathcal{NP}$-complete problem.

2.9 Listing (Enumeration) of Problems

We would like to be able to list *all* of the problems in a class. For example, we can list the $\mathcal{RE}$ problems by listing the Turing machines. Each Turing machines then represents the set of inputs which cause that particular machine to eventually **HALT**.

2.9.1 Clocked Machines

OOPS! better include or explain that the clocks have to be primitive recursive computable.

All PRIM can be listed by PRIM programs.

All PRIM can be listed by clocked PRIM programs.

All PRIM can be listed by clocked programs. All PRIM can be listed by clocked Turing machines.

All P can be listed by polynomial clocked Turing machines.

All NP can be listed by polynomial clocked non-deterministic Turing machines.

NP inter coNP is a lot harder.
We need both a poly time acceptor and a poly time rejector.
AND, we need this pair of machines to be complete and consistent, in the sense that exactly one of the machines HALTS on each input. (Complete means that *at least one* machine halts for each input.) (Consistent means that the two machines *never* give conflicting answers.)

All NP can be listed by polynomial clocked non-deterministic Turing machines.

2.10 A Typology for Reductions

One reduction relation is not sufficient for all purposes. We may need to limit the resources allowed in the transformation of an instance of one problem to an instance of another problem. Further, we may want to limit how we can ask questions about the problem we are reducing to. In this section, we want to discuss these two issues – limited transformation and limited questioning. While we can devise a typology for various kinds of reduction, we will still have to admit that there are many questions about the relationships between the various notions of reducibility which have not been satisfactorily answered.

[INCLUDE Turing Time Restricted
Many-One Space Restricted
One-to One
Truth Table Other Restrictions]

2.10.1 Extremely Simple Reductions

2.11 Finite State Reductions

The usual simplest complexity class is the **Regular** languages which are the set of strings which are recognized by **finite automata**. There is a special subclass of these languages that is worth mentioning, namely the *Finite/coFinite* languages. **Finite/coFinite** languages are the sets of strings which either have a finite number of strings or are the complements of sets with a finite number of strings. This class is almost the same as the class of languages recognized by neural nets with no loops in their connections, e.g. the graph of the connection diagram of such a neural net is a directed acyclic graph (DAG). The slight difference between loop-free and Finite/coFinite is that the loop-free nets cannot remember what happened in time instants which are longer ago than the depth of the net. So, the set of strings accepted by a loop-free net can be written in the form $\Sigma^* F$ where Σ is the input alphabet and F is a Finite or coFinite set.

We want to consider the appropriate reduction for these sets. Above we argued that CFL, the class of context free languages, has a hardest language and we showed that this language L_0 is a complete language for CFL under reduction using homomorphisms. Here, we want to argue that homomorphisms are a bad choice, because they do not make all regualar languages equivalent. And in fact, homomorphisms **MAY** even allow Finite to be incomparable.

2.11.1 Finite State Mappings

A machine takes as input a string over some alphabet and outputs a string over the same or a different alphabet. We want to discuss the string transformations which are possible when the machine has a finite memory. (We are modeling a real computer with its finite internal memory. Unbounded external memory is not included in this model.) A machine with this memory limitation is usually called a **finite state machine**. Such a machine is described by specifying the input and output alphabet, the finite number of states, and two functions – the next state function and the output function.

The **next state function** calculates the new state given the present state and the present input symbol. This can be symbolized as

$$q_{t+1} \;=\; F(\, q_t,\, s_t\,)$$

where q_{t+1} is the state at time $t+1$ and it is determined from q_t, the state at time t, and s_t, the input symbol at time t. We call $F(q,t)$ the *next state function.*

The output of the machine could depend on the present state or on the present state and the input symbol. It turns out that either of these two choices for the output function will lead to essentially the same mappings and so we will choose the slightly simpler convention that the output is only a function of the present state. A machine with this output convention is called a *Moore machine.* Such machines are usually allowed to have only one output symbol for each time step, but we will find it more convenient to allow at output string for each time step. Symbolically, we write

$$w_t \;=\; G(\,q_t\,)$$

and mean that if the machine is in state q_t at time t, then it outputs the string w_t

At the moment, both F and G deal with one time step at a time. We want to extend these functions so that they become functions of the whole input string. To do so, we define

$$F^*(\,q,\,x\,) \;=\; \begin{cases} q & \text{if} \quad x = \Lambda \text{ (Null string)} \\ q\;F^*(\,F(q,\,s),\,y\,) & \text{if} \quad x = sy \end{cases}$$

where s is a single input symbol and x and y are input strings. With this definition, F^* gives us a mapping from input strings to sequences of states In fact, F^* gives us several mappings, one for each state q. If we specify one state, say q_0, as the initial state, then $F^*(\,q_0,\,x\,)$ is the state sequence mapping for the machine, and it maps an input string $x = x_1, \ldots, x_n$ to a sequence $\langle q_0, q_1, \ldots, q_n\rangle$ of states. Since the output function maps each state to an output sequence, the overall output produced by the input string x is

$$G(\,q_0\,)\cdot G(\,q_1\,)\cdot \cdots \cdot G(\,q_n\,)$$

where $G(\,q_i\,)$ is the output string corresponding to the state q_i and $\cdot$ indicates that all of these strings are being concatenated together. We can represent this entire concatenated string symbolically as

$$G^*(\,x\,)$$

and think of G^* as mapping input strings to output strings. Notice that G^* does not map strings of length n to strings of length n. Because $G(\,q\,)$ is a string and may possibly be the null string, the concatenation of $n+1$ such strings may produce a string shorter or longer than n characters. Of course, if we knew $K \;=\; \max_q |\,G(\,q\,)\,|$ then we could be sure that

$$|\,G^*(\,x\,)\,| \;\leq\; K\,(\,|x| + 1\,).$$

2.12 $\mathcal{NP}$-Reductions

Reductions do not have to be deterministic, they can be non-deterministic. For example, we can define a reduction based on $\mathcal{NP}$, that is a non-deterministic polynomial time reduction. If we define this reduction reasonably, we expect that all (non-trivial) $\mathcal{NP}$ problems will be equivalent with respect to this reduction.

Define

$$A \ \leq_{\mathcal{NP}} \ B$$

if and only if there is a polynomial $p(x)$ and a function of two variables $g(y, z)$ so that

$$\begin{array}{lll} \forall I \in A & \exists J \text{ with } |J| \leq p(|I|) & g(\,I,\,J\,) \in B \\ \forall I \notin A & \forall J \text{ with } |J| \leq p(|I|) & g(\,I,\,J\,) \notin B \end{array}$$

and

$$T_g(\,|I|,\,|J|\,) \ \leq \ \text{poly}(|I|).$$

(That is, the running time of g is bounded by a polynomial in the size of I.)

If B is any (non-trivial) $\mathcal{NP}$-set and A is any $\mathcal{NP}$-set then

$$g(\,I,\,J\,) \ = \ \begin{cases} b_{YES} & \text{if } \exists J \ \ v_A(I,J) \ = \ \text{YES} \qquad (b_{YES} \in B) \\ b_{NO} & \textbf{ELSE} \qquad\qquad\qquad\qquad\quad (b_{NO} \notin B) \end{cases}$$

where v_A is the acceptor for A, i.e. $I \in A$ iff $\exists J$ so that $v_A(I,J) = \text{YES}$, and the time to evaluate $v_A(I,J)$ is bounded above by a polynomial in $|I|$.