Chapter 1 : Introduction

Complexity theory attempts to prove that some computational problems are more complex than some other problems. Here 'complex' typically refers to (but is not restricted to) time and/or space requirements for solving a problem. One defines some synthetic complexity classes and populates those classes with common computational problems. The theory deals with proving relationships (like containment, strict or otherwise) among these classes. The motivation is that the smallest class in which a given problem resides offers interesting and useful theoretical guidelines regarding the practical difficulty of solving that problem using *computers*.

But then what is a (computational) problem? Intuitively speaking, it is a *problem* that we plan to solve by computers, like searching databases, finding paths in a graph, a smart chess program and so on. I won't bother to define problems more rigorously. There is seemingly no danger caused by this laziness (or inability). You need not define everything under the sun, or for that matter the sun itself!

A more relevant question is: what is a computer? It is a computing device, what else? But what is computation? A computation is a well-specified sequence of instructions carried out by a computing device. Wow! I am getting entangled in a loop, but I will come out soon. It is required for us to propound rigorous definitions of computers.

There is little harm, if you call your PC or work-station a prototype of a computer and whatever one can solve with it in a *finite* amount of time a computational problem. However, it is customary and useful, at least pedagogically, to define our own abstract and yet concrete computing device. This is what I will do next and will argue that under reasonable assumptions this abstract machine is as *powerful* as your PC or work-station.

1.1 Turing machines

A finite set is called an alphabet. A string over an alphabet Σ is a finite (ordered) sequence of elements of Σ. The set of all strings over Σ is denoted by Σ^*. The empty string is denoted by ϵ. The length $|\alpha|$ of a string $\alpha \in \Sigma^*$ is the number of symbols (counted with repetitions) in α. Thus $|\epsilon| = 0$.

Any subset $L \subseteq \Sigma^*$ is called a language over Σ. We intend to have finite representations of languages. If $L \subseteq \Sigma^*$ is already finite, we can simply enumerate all the members of L one by one. If L is infinite, this strategy does not work and we require separate machineries to achieve this goal. In your FLAT course, you have seen many such schemes: regular grammars, context-free grammars, context-sensitive grammars and unrestricted grammars (finitely) represent bigger and bigger classes of languages. Also we have computing devices that can represent such languages – finite automata represent regular languages, pushdown automata represent context-free languages and Turing machines represent recursively enumerable languages.

Given a finite representation of $L \subseteq \Sigma^*$ and a string $\alpha \in \Sigma^*$, we face the question of deciding if $\alpha \in L$. This language membership problem is the central problem in the theory of computation. It is essentially a decision problem having only a "yes" or "no" answer. Many interesting problems we plan to solve by computers can be rephrased in terms of some language membership problem, some others cannot be. However, viewing each computational problem as a language membership problem, though expressive of our inability to represent problems in a better way, has stood the taste of time.

Simple counting arguments show that we can have finite representations of only countably infinite languages, whereas the number of languages is uncountable. Thus our computing devices can recognize much less languages than actually exist. We have to accept this bad news.

A Turing machine (TM) has a finite control that can at any time be in one of a finite set Q of states. It also contains a tape extending infinitely in both directions and divided into (finite) cells. Each tape cell is capable of holding one of a finite set of symbols, called the tape alphabet Γ. The machine also comes with a read/write head that can scan and write symbols at the tape cells. The machine starts its operation with the following configuration:

- The input string (over some alphabet Σ) is written on the tape and every other tape cell holds a special blank symbol $\sqcup$. It is assumed that $\Sigma \subseteq \Gamma$ and that $\sqcup \notin \Sigma$.
- The read/write head is positioned at the first symbol of the input.
- The finite control is in a distinguished start state s.

The machine then enters a loop consisting of the following steps. The read/write head scans the tape symbol $a \in \Gamma$ from the cell pointed to by it. Depending on this symbol a and the current state $q \in Q$ of the finite control, the machine undergoes a transition $\delta(q, a) = (p, b, D)$, where:

- $p \in Q$ is the new state of the finite control (one may have $p = q$),
- the read/write head overwrites the tape symbol a by b (one may have $a = b$), and
- the head moves to the next tape cell, either to the left ($D = L$) or to the right ($D = R$) of the current position.

There is a distinguished subset $F \subseteq Q$, called the set of final or accepting states. If the new state $p \in F$, the machine *accepts* the input string and halts. The transition $\delta(q, a)$ may be undefined for some combinations of q and a. In that case, if $q \notin F$, then the machine *rejects* the input string and halts. The machine may also enter an endless loop without ever reaching a final state. In this case too, we say that the input string is rejected.

1.1 Definition A Turing machine M is formally defined as a tuple $(\Sigma, \Gamma, \sqcup, Q, s, F, \delta)$ with its elements Σ through δ interpreted as above. This is clearly a finite representation. The infiniteness of the tape should not bother you. The language of a TM M is defined to be the set

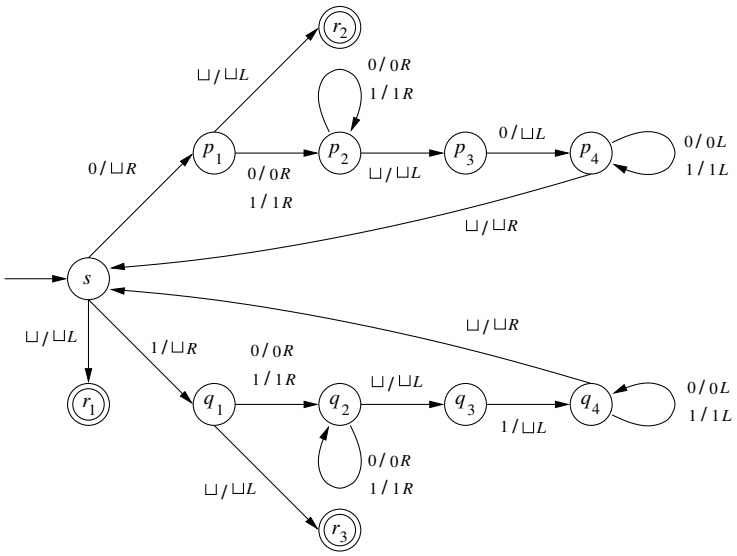
$$\mathcal{L}(M) = \{\alpha \in \Sigma^* \mid M \text{ accepts (and halts) for input } \alpha\}.$$

We say that M has (represents) the language $\mathcal{L}(M)$, i.e., M is intended to solve the membership problem for the language $\mathcal{L}(M)$. A language accepted by a Turing machine is often called a recursively enumerable or an r.e. language.

1.2 Example Let us construct a Turing machine M that accepts all (and only) the palindromes over the binary alphabet $\Sigma := \{0, 1\}$. We take $\Gamma := \{0, 1, \sqcup\}$ and $Q := \{s, p_1, p_2, p_3, p_4, q_1, q_2, q_3, q_4, r_1, r_2, r_3\}$, where s is the start state and the final states are r_1, r_2, r_3. The transitions are as shown in the Figure 1.1. Informally, M reads the leftmost symbol, remembers it, deletes it and then tries to find the same symbol at the right end. If the input is a palindrome of even length, M eventually reaches the accepting state r_1. If the input is a palindrome of odd length, the machine accepts by entering either state r_2 or r_3. Finally, if the input is not a palindrome, it will be eventually detected in either state p_3 or state q_3, and M will halt without accepting. Thus M halts on all inputs. This is, however, not a mandatory behavior on the part of a Turing machine.

It is interesting to look at the number of elementary steps (transitions) needed by M on an input string of length n. Each detection of a match requires a complete forward march and a complete backward march through the remaining portion of the string. Thus the total number of steps is no more than (roughly) $2\Big(n + (n-2) + (n-4) + \cdots\Big)$ which is $\mathrm{O}(n^2)$. Also note that M uses only a single additional cell after the end of the input.

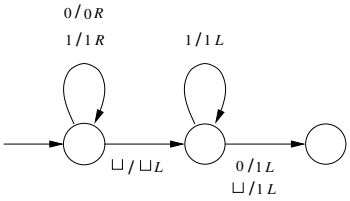
Figure 1.1: A Turing machine that accepts binary palindromes



Detecting palindromicity is thus an *easy* problem. How would you do it in your PC using a high-level language? In what time?

1.3 Example A TM can also be used to *do something*, i.e., to compute a function $f : \Sigma^* \rightarrow \Gamma^*$. This carries the implication that if α is the input, then the machine halts with $f(\alpha)$ (and nothing else) on its tape. As an example, consider a TM that takes a binary string α as input and halts after computing $\alpha + 1$ in its tape. Here binary strings are identified with (non-negative) integers in the natural way (with ϵ identified with 0). Figure 1.2 depicts the transition diagram for a simple TM that performs this increment operation. No final states are marked, since the machine is not meant for accepting a language in the conventional sense. It runs in $O(n)$ time for an n-bit input integer.

Figure 1.2: A Turing machine that increments a natural number



So far so good! But how robust and general is this model? Some obvious enhancements of the basic model immediately come to our mind. For example, we may add other (finite or infinite) tapes, each having its own read/write head or we may even allow multiple heads per tape. One may also think of multi-dimensional tapes, registers (finite memory) in the control, multiple tracks in a tape, facility for a head to remain stationary during a transition, and so on. It turns out that the basic model can simulate all these enhanced versions – the only cost one may have to pay is an increased running time for the simulated machine. But it is vital to note here that if a problem is solvable in polynomial time by an enhanced machine, it is also solvable in polynomial time by a simulator, though the degree of the polynomial may be higher for the basic machine (Exercises 1.1.1, 1.1.2). These observations have important impacts while we define complexity classes, as we will see later.

Finally, how does a TM compare with a computer? A computer possesses a *random-access memory*, whereas the memory of a TM (i.e., its tape) is always scanned sequentially – arbitrary jumps are not allowed. On the other hand, we have compelling evidence to believe that the universe is *finite*, and so a computer can

never simulate the infinite tape of a TM and is essentially a finite automaton. But if we use removable media (like CDROM) to simulate the tape and assume that we will get as many units of the medium as we would feasibly desire, the simulation is practically perfect. After all, our universe is pretty big and has enough matter in it to manufacture astronomically large memory. Under these assumptions it is not difficult to establish the computational equivalence of a TM and a computer. More interestingly, this equivalence preserves polynomial behavior (running time and space) – only the polynomial degrees may be affected.

Exercises for Section 1.1

1. Consider the language L of binary palindromes discussed in Example 1.2. Show that a two-tape TM M can be designed with $\mathcal{L}(M) = L$ such that M requires at most $\mathrm{O}(n)$ transitions for an input string of length n.

2. Consider the language $L := \{0^k 1^k \mid k = 0, 1, 2, 3, \ldots\} \subseteq \{0, 1\}^*$.

(a) Design a single-tape single-head TM to accept L in $\mathrm{O}(n^2)$ time (where n is the length of the input).

*** (b)** Design a single-tape single-head TM to accept L in $\mathrm{O}(n \log n)$ time.

(c) Design a two-tape TM to accept L in $\mathrm{O}(n)$ time.

(d) Can you do it in less than $\mathrm{O}(n)$ time using a TM?

3. Let M be a TM for which every transition $\delta(q, a)$, if defined, is of the form (p, b, R), i.e., M never moves it head left. Show that $\mathcal{L}(M)$ is regular.

4. Let L be an RE language over the alphabet Σ. Show that L is the language of a TM M with the property that M does not halt on some input, if and only if $L \neq \Sigma^*$.

1.2 Nondeterminism

So far we have talked about the deterministic version of a Turing machine, often called a deterministic Turing machine (DTM). For a state q and a scanned symbol a the transition $\delta(q, a)$ for a DTM is either undefined or has a *unique* value (p, b, D). In other words, for each (q, a)-combination the DTM has a single instruction of either halting or a unique transition.

For a nondeterministic Turing machine (NTM) we allow multiple choices for each (q, a)-combination. That is, we now have $\delta(q, a) = \{(p_1, b_1, D_1), (p_2, b_2, D_2), \ldots, (p_k, b_k, D_k)\}$ for some $k \geqslant 0$. If $k = 0$, the NTM halts. If $k = 1$, the machine carries out a unique transition. If $k \geqslant 2$, what would the machine do? Is there a question of guessing, or selecting a *good* move, or randomly choosing a move? The answer is that the NTM will do nothing of these above possibilities. In fact, it will make all the transitions *simultaneously*.

Note that a DTM is clearly an NTM by definition, but the converse is not necessarily true. In future, an unspecified use of the term TM will imply a DTM. Only when it is imperative to highlight the role of determinism, we will use the term DTM.

You must be thinking of a C program now. Yes, in C you can create multiple processes for different choices. You may also have a single-process implementation using backtracking, or you may even explore all the possibilities one by one. There is a catch in the last two options. You may get lost in an endless loop for a particular choice and never get the opportunity to explore other alternatives. This problem can be avoided by choosing the moves in a suitable order, as we will discuss a little later. For the time being, let us investigate what extra power we add to the machine by giving it multiple choices during a transition.

We say that an NTM accepts an input string, if (and only if) there is (at least) one sequence of choices of the transitions, that eventually leads the machine to an accepting state. The machine rejects the input, if each possible choice leads to either halting in a non-final state or to an endless loop. The language of an NTM is the set of all input strings that the machine accepts (nondeterministically, i.e., in the sense just explained).

It is not known *a priori* what opportune choices would make an NTM accept a string in its language. But this issue arises only when one wants to simulate the behavior of an NTM in an otherwise deterministic system, like a PC or a DTM. That way we arrive at a conclusion if nondeterminism leads to more power in recognizing languages. You have seen in your FLAT course that nondeterminism does not increase the language accepting power for a finite automaton, but in many cases it helps us have a more compact representation (i.e., a representation with fewer — sometimes exponentially fewer — states). In connection with pushdown automata, nondeterminism adds to the language-recognition power, i.e., the class of languages accepted by nondeterministic PDA is a strict superset of the class of languages accepted by deterministic PDA. What do we now gain in the scenario of TMs?

An NTM can be simulated in a DTM, as you have seen in your FLAT course. Thus NTMs accept only as many languages as the DTMs do. The simulation is very important for complexity theory and so let me spend some time recapitulating the procedure. Recall that the c o n f i g u r a t i o n of a DTM or NTM at a point of time during its computation is specified by its state, the content of its tape and the position of its head in the tape. Note that the tape is infinite, but at any instant of time the tape has only finitely many cells occupied by non-blank symbols. If existence of infinite sequences of blank symbols on either side of the non-blank portion is implicitly assumed, the configuration then can be identified as a finite quantity. A possibility is exemplified by: $q, abc\underline{a}bc$ or $q, abc\underline{a}\sqcup b\sqcup\sqcup c$. Here the underline represents the position of the head. In certain cases one may have to include a finite portion of one of the (semi-infinite) blank portions of the tape. Here are examples: $\underline{\sqcup}\sqcup\sqcup abc$ or $abc\sqcup a\sqcup b\sqcup c\sqcup\underline{\sqcup}$. Since a DTM or NTM has only finitely many states, a configuration of a machine can be encoded by a finite string of symbols. We can easily conceive of a DTM that understands the encoding of the configuration of (another) DTM or NTM.

Now let N be an NTM and we plan to build a DTM M that simulates the behavior of N in such a way that M accepts an input α (deterministically) if and only if N accepts α nondeterministically. The idea is to explore configurations of N in M's tape in a breadth-first search fashion. M's finite control embeds the knowledge of the transitions of N.

Suppose that M is asked to simulate N's behavior on an input α. M starts its operation with α on its tape (and with N in its head). M knows that initially N is in its (N's) start state and N's read/write head points to the first symbol of the input. Thus it is simple for M to convert α to the initial configuration for N on α. It is expedient to view M's tape as a queue of M's configuration. To start with M pushes the initial configuration of N to an empty queue.

Subsequently, M carries out the following loop, until the queue in its tape is empty. At the front of its queue M locates an encoded configuration of N. From this M knows N's state (say, q), the content of N's tape and the precise position of N's head on the tape, i.e., the symbol (call it a) that N is currently scanning. If q is a final state for N, M accepts and halts. Otherwise, M employs its knowledge of N's transition table to obtain $\delta_N(q, a) = \{(p_1, a_1, D_1), (p_2, b_2, D_2), \ldots, (p_k, b_k, D_k)\}$. For each $i \in \{1, \ldots, k\}$ M computes the next configuration of N for the choice (p_i, b_i, D_i) and inserts the configuration at the rear of the queue on M's tape. In order to separate two consecutive configurations in the queue M may use a delimiter, a symbol that does not appear in any encoding of N's configurations. M then deletes the previous configuration that it processed in this execution of the loop and goes back to the top of the loop, provided the queue still contains configurations to be explored. If the queue is empty, M rejects and halts.

It is not difficult to establish that this simulation lets M accept α eventually, if and only if N does so. Thus any language accepted by an NTM is also accepted by a DTM. The above simulation is machine-specific, i.e., we have a DTM for each NTM. A universal simulator can likewise be prepared, but that's not an immediate concern to us. Since a DTM is also an NTM, we come to the conclusion that the class of languages accepted by NTMs is precisely the class of languages accepted by DTMs.

But the story does not end here. It begins here!!! Assume that m is the fan-out of N (i.e., the maximum number of choices in one transition of N). Suppose also that there is a sequence of n transitions of N that leads to the acceptance of α. Thus if N makes a good guess all the time, it can accept α in no more than n

moves. In order to reveal this complete sequence, M may have to explore as many as $\mathrm{O}(m^n)$ configurations of N. If $m \geqslant 2$, this leads to an exponential increase in the running time.

But is this exponential slow-down unavoidable? In other words, does there exist a speedier simulation? M does not know *a priori* a sequence of good guesses that may lead to acceptance, and so it *has* to explore all possibilities. Informally, this convinces us of a negative answer to the question I mentioned at the beginning of this paragraph.

Look at the language $\mathcal{L}(M) = \mathcal{L}(N)$. A straightforward simulation of N by M may lead to exponential blow up, but this never rules out the possibility that a DTM M' (other than M) cannot be designed with $\mathcal{L}(M') = \mathcal{L}(M)$. That M' need not at all be a simulation of N. In complexity theory problems are important, not the particular ways of solving them. This important observation leads to the famous $\mathrm{P} = \mathrm{NP}$? question. We still don't have an answer!

Exercises for Section 1.2

1. Consider the language $L := \{\alpha\alpha \mid \alpha \in \Sigma^*\} \subseteq \Sigma^*$, where $\Sigma = \{a, b\}$.

(a) Design a one-tape one-head DTM that accepts L in $\mathrm{O}(n^2)$ time (where n is the length of the input string).

(b) Design a one-tape two-head DTM that accepts L in $\mathrm{O}(n)$ time.

(c) Design an NTM that accepts L in $\mathrm{O}(n)$ time. (You may use a scratch tape.)

2. Establish that an arithmetic operation (comparison, addition, subtraction, multiplication or Euclidean division) can be carried out on a DTM in polynomial time (of the input size).

3. Let $L := \{\alpha \in \{0,1\}^* \mid 1\alpha \text{ is composite }\}$, where strings are naturally identified with non-negative integers.

(a) Design an NTM N with $\mathcal{L}(N) = L$ and that accepts a string in polynomial time. The machine must halt (in a nondeterministic sense), if 1α happens to be a composite.

(b) Can the NTM you designed in Part (a) be used in the same way to check if an input integer is prime?

*** 4.** Let $k \geqslant 3$ and N a k-tape NTM with $L := \mathcal{L}(N)$. Suppose that N accepts an input of size n in nondeterministic time $f(n)$. Design a two-tape NTM N' with $\mathcal{L}(N') = L$, such that N' also accepts strings of size n in the same nondeterministic time $f(n)$. (Assume that each tape has only one read/write head.)

1.3 Undecidability

I have already mentioned that anything under the sun is not computable. Now we have a class of powerful machines (namely the Turing machines), that solves a fairly big collection of problems. But Turing machines have got a problem. It need not halt for some input(s). We, however, demand that every decision, "accept" or "reject", must come in a *finite* amount of time. This finite time may be astronomically big, may be even bigger than the age of the universe, but that's still better than waiting forever to know the machine's decision.

1.4 Definition An algorithm is formally defined to be a TM that halts on every input. The language of such a TM is called recursive.

The language of any TM is called recursively enumerable (r.e.), as I mentioned earlier. It follows that any recursive language is r.e. too. But what about the converse? It turns out that there are r.e. languages that are not recursive. You have seen some examples in the FLAT course, like the universal language L_u (the language of the universal Turing machine). There are also (there have to be) languages that are not even recognized by any TM, like the diagonalization language L_d. Locating such languages is somewhat difficult, but doable.

In complexity theory problems are important, not some algorithms to solve them. But whenever we talk about an algorithm, we require it to halt eventually. Thus we must concentrate only on recursive problems, also known as decidable problems. Anything beyond that is of little concern to complexity theory.

But what is the motivation behind Definition 1.4. What is so divine about TMs that always halt, that leads to its acceptance as the formal definition if an algorithm? The answer is historical. In 1900 the eminent mathematician David Hilbert posed some twenty odd open problems of mathematics. In the tenth problem he asked about finding an effective procedure to determine the number of zeros of a multi-variate polynomial. Mankind didn't have devised computers at that time, but the question of *effective computability* bothered the twentieth century mathematicians. Several researchers came up with different notions of computability:

- Type 0 (or unrestricted) grammars
- Theory of recursive functions
- λ-calculus
- Post systems
- Combinatory logic
- Turing machines

Nowadays we may even add a sufficiently powerful programming language (like C) to this list. It was discovered that however diverse these different notions looked, they eventually addressed the *same* class of problems. This universality of these different models led to the following conjecture.

1.5 Conjecture [Church-Turing thesis] The best conceivable (i.e., widest) notion of an algorithm is that propounded by Definition 1.4.

This means that we cannot design a class of machines (with finite representations) that can accept (with halting) a bigger class of languages, than the TMs do. It's an unprovable conjecture, because it attempts to relate a mathematical object (TM) with a non-mathematical one (algorithm). One can, however, disprove the conjecture by designing a better class of computing devices. But after about seven decades of the proposal of the conjecture, no such machine could be conceptualized. Many tend to believe that the recursive languages constitute the limit which we cannot transcend with our present-day knowledge of logic.

Now that our machines halt on every input, it's time to give a precise definition of the running time of an algorithm. Instead of concentrating on specific inputs, we parameterize running time in terms of the input size. More often than not, we take interest in the worst-case running time.

1.6 Definition The running time of a deterministic algorithm for an input of size n is the maximum number of transitions taken by the algorithm before halting, where the maximum is taken over all possible strings of size n (irrespective of whether the string is accepted or rejected). The running time of a nondeterministic algorithm for an input of size n is the maximum number of transitions in a branch of execution (from the start to a halt configuration), where the maximum is taken over all possible branches of execution for all possible inputs.

Exercises for Section 1.3

1. Show that:

(a) R.E. languages are closed under (set-theoretic) union and intersection.

(b) Recursive languages are closed under complement.

(c) If L and $\bar{L}$ are both r.e., then L (and hence $\bar{L}$) are also recursive.

2. We define the Kleene closure or Kleene star of a language $L \subseteq \Sigma^*$ by

$$L^* := \{\alpha_1 \dots \alpha_k \mid k \geqslant 0 \text{ and each } \alpha_i \in L\}.$$

*** (a)** Show that if L has a poly-time deterministic algorithm, then so does L^*.

(b) Show that if L has a poly-time nondeterministic algorithm, then so does L^*.

1.4 Problems and reductions

The basic difference between the study of algorithms and complexity theory lies in that in the former, one designs an algorithm to solve a particular problem and analyzes the complexity (time or space) of that algorithm. Better complexity figures mean better algorithms. In complexity theory, on the other hand, we talk about the complexity of a problem and not of a set of particular algorithms to solve that problem. The difficulty with this goal is that it is impractical, if possible at all, to enumerate all algorithms to solve a problem and take the best complexity of these algorithms as the complexity of the problem. (There may be infinitely many algorithms that solve the same problem.) But this inability need not discourage us from investigating complexity results about problems. Certain assertions can still be proved.

One important tool to deal with the *relative complexity* of two problems is r e d u c t i o n between problems. Let L_1 and L_2 be two languages over alphabets Σ_1 and Σ_2 respectively. A reduction from P_1 to P_2 is an algorithm that given any $\alpha \in \Sigma_1^*$ halts with $f(\alpha) \in \Sigma_2^*$ such that $\alpha \in L_1$ if and only if $f(\alpha) \in L_2$. If there is a reduction from L_1 to L_2, we write $L_1 \leqslant L_2$. The implication is that if $L_1 \leqslant L_2$ and we have an algorithm M_2 for solving L_2, we also have an algorithm for L_1 too, i.e., just convert, by the reduction algorithm, the input α for L_1 to the input $f(\alpha)$ for L_2 and run M_2 on $f(\alpha)$ and accept the decision (accept/reject) of M_2 to be the decision for L_1. It, however, may be the case the L_1 has a speedier algorithm than the reduction-followed-by-run-M_2 algorithm.

Recall the following joke. A mathematician is asked to put a bucket full of water on a table. He enters a room having a table, a sink with a water tap and an empty bucket on the floor. The mathematician takes the bucket to the sink, fills it up with water and puts the filled bucket on the top of the table. Another day, he is asked to do the same thing. When he entered the room, he found the table and the sink as they were, but the bucket was already full of water and standing on the floor. He thought for a while, then took the bucket to the sink, carefully cleared its contents in the sink, puts back the resulting empty bucket on the floor and sighs with relief that he has already reduced the new problem to the old problem which he can solve.

That's reduction. If every problem in a complexity class reduces to a particular problem P, then P is (one of) the most difficult problems in the class. This is how we define c o m p l e t e p r o b l e m s for a complexity class. If a complete problem is easy to solve, then every other problem in the class is also easy.

There is a catch in the above discussion. We never talked about the complexity of the reduction algorithm. Think of $P_1 \leqslant P_2$ and P_2 having a poly-time algorithm. If the reduction algorithm is super-poly-time (say, exponential-time), then reduction gives a super-poly-time algorithm for P_2 and does not prove that P_2 is at least as difficult as P_1. When we do reductions in a complexity class, we must, therefore, have reduction algorithms that are easier than or at most as difficult as the algorithms in the class. Thus for the class P we talk about poly-time reductions, not any arbitrary reduction. The notion of reduction varies with the context and must be seriously understood before we make an attempt to apply them.

Exercises for Section 1.4

1. Let $G = (V, E)$ be an undirected simple graph and assume a *reasonable* encoding of the graph, say, using the adjacency matrix. Devise poly-time algorithms for the following problems:

(a) Given (the encoding for) G and two vertices $u, v \in V(G)$, determine if v is reachable from u.

(b) Given G, determine if G contains a cycle.

(c) Given G, determine if G has a cycle of odd length.

(d) Given G, determine if G is bipartite.