26. Devise a parallel algorithm which computes the set of values $x^2, x^3, \ldots, x^n$ which requires less than $O(n)$ time.

27. [Kung] Consider the recurrence relation $y_{i+1} = (1/2)(y_i + a/y_i)$ $i = 0, 1, 2, \ldots, n-1$ for approximating $a^{1/2}$. Show that evaluating y_n by any parallel algorithm requires $O(n)$ parallel time.

28. [Kung] Given the recurrence $y_i = y_{i-1}b_i + a_{i+1}$, $i \geq 1$, show that a speed-up of at most $(2/3)k + 1/3$ is the best possible for evaluating y_n.

29. [Borodin Munro] This exercise completes the proof of Theorem 10.9. Let $p_1(a_1, \ldots a_s), \ldots, p_u(a_1, \ldots, a_s)$ be u linearly independent functions of $a_1, \ldots, a_s$. Let $a_1 = p(a_2, \ldots, a_s)$. Then show that there are at lest $u - 1$ linearly independent $p_i' = p_i$ where a_1 is replaced by p.

30. Devise a parallel algorithm which computes the value of an nth degree polynomial in time $O(\log n)$.

31. Devise a parallel algorithm which merges two ordered sets of n elements in $O(\log n)$ time.

32. [W. Miller] Show that the inner product of two n-vectors can be computed in $\lceil n/2 \rceil$ multiplications if separate preconditioning of the vector elements is not counted.

FROM: FUNDAMENTALS OF COMPUTER ALGORITHMS by HOROWITZ & SAHNI (2)

Chapter 11

NP-HARD AND NP-COMPLETE PROBLEMS

11.1 BASIC CONCEPTS

This chapter contains what is perhaps the most important theoretical development in algorithms research in the past decade. Its importance arises from the fact that the results have meaning for all researchers who are developing computer algorithms, not only computer scientists but electrical engineers, operations researchers, etc. Thus we believe that many people will turn immediately to this chapter. In recognition of this we have tried to make the chapter self-contained. Also, we have organized the later sections according to different areas of interest.

There are however some basic ideas which one should be familiar with before reading on. The first is the idea of analyzing apriori the computing time of an algorithm by studying the frequency of execution of its statements given various sets of data. A second notion is the concept of the order of magnitude of the time complexity of an algorithm and its expression by asymptotic notation. If $T(n)$ is the time for an algorithm on n inputs, then, we write $T(n) = O(f(n))$ to mean that the time is bounded *above* by the function $f(n)$, and $T(n) = \Omega(g(n))$ to mean that the time is bounded *below* by the function $g(n)$. Precise definitions and greater elaboration of these ideas can be found in Section 1.4.

Another important idea is the distinction between problems whose solution is by a polynomial time algorithm ($f(n)$ is a polynomial) and problems for which no polynomial time algorithm is known ($g(n)$ is larger than any polynomial). It is an unexplained phenomena that for many of the problems we know and study, the best algorithms for their solution have computing times which cluster into two groups. The first group consists of problems whose solution is bounded by a polynomial of small degree. Examples we have seen in this book include ordered searching which is $O(\log n)$, poly-

nom... evaluation is $O(n)$, sorting is $O(n \log n)$, and matrix multiplication which is $O(n^{2.81})$.

The second group contains problems whose best known algorithms are nonpolynomial. Examples we have seen include the traveling salesperson and the knapsack problem for which the best algorithms given in this text have a complexity $O(n^2 2^n)$ and $O(2^{n/2})$ respectively. In the quest to develop efficient algorithms, no one has been able to develop a polynomial time algorithm for any problem in the second group. This is very important because algorithms whose computing time is greater than polynomial (typically the time is exponential) very quickly require such vast amounts of time to execute that even moderate size problems cannot be solved. (See Section 1.4 for more details.)

The theory of NP-completeness which we present here does not provide a method of obtaining polynomial time algorithms for problems in the second group. Nor does it say that algorithms of this complexity do not exist. Instead, what we shall do is show that many of the problems for which there is no known polynomial time algorithm are computationally related. In fact, we shall establish two classes of problems. These will be given the names NP-hard and NP-complete. A problem which is NP-complete will have the property that it can be solved in polynomial time iff all other NP-complete problems can also be solved in polynomial time. If an NP-hard problem can be solved in polynomial time then all NP-complete problems can be solved in polynomial time. As we shall see all NP-complete problems are NP-hard but all NP-hard problems are not NP-complete.

While one can define many distinct problem classes having the properties stated above for the NP-hard and NP-complete classes, the classes we study are related to nondeterministic computations (to be defined later). The relationship of these classes to nondeterministic computations together with the "apparent" power of nondeterminism leads to the "intuitive" (though as yet unproved) conclusion that no NP-complete or NP-hard problem is polynomially solvable.

We shall see that the class of NP-hard problems (and the subclass of NP-complete problems) is very rich as it contains many interesting problems from a wide variety of disciplines. First, we formalize the preceding discussion of the classes.

Nondeterministic Algorithms

Up to now the notion of algorithm that we have been using has the property that the result of every operation is uniquely defined. Algorithms with this property are termed *deterministic algorithms*. Such al... hms agree with the way programs are executed on a computer. In a theoretical framework we can remove this restriction on the outcome of every operation. We can allow algorithms to contain operations whose outcome is not uniquely defined but is limited to a specified set of possibilities. The machine executing such operations is allowed to choose any one of these outcomes subject to a termination condition to be defined later. This leads to the concept of a *nondeterministic algorithm*. To specify such algorithms we introduce one new function and two new statements into SPARKS:

(i) **choice** (S) . . . arbitrarily chooses one of the elements of set S
(ii) **failure** . . . signals an unsuccessful completion
(iii) **success** . . . signals a successful completion.

The assignment statement $X \leftarrow$ **choice**$(1{:}n)$ could result in X being assigned any one of the integers in the range $[1, n]$. There is no rule specifying how this choice is to be made. The **failure** and **success** signals are used to define a computation of the algorithm. These statements are equivalent to a **stop** statement and cannot be used to effect a **return**. Whenever there is a set of choices that leads to a successful completion then one such set of choices is always made and the algorithm terminates successfully. *A nondeterministic algorithm terminates unsuccessfuly if and only if there exists no set of choices leading to a success signal*. The computing times for **choice**, **success**, and **failure** are taken to be $O(1)$. A machine capable of executing a nondeterministic algorithm in this way is called a *nondeterministic machine*. While nondeterministic machines (as defined here) do not exist in practice, we shall see that they will provide strong intuitive reasons to conclude that certain problems cannot be solved by "fast" deterministic algorithms.

Example 11.1 Consider the problem of searching for an element x in a given set of elements $A(1{:}n)$, $n \geq 1$. We are required to determine an index j such that $A(j) = x$ or $j = 0$ if x is not in A. A nondeterministic algorithm for this is

```
j ← choice(1:n)
if A(j) = x then print(j); success endif
print('0'); failure
```

From the way a nondeterministic computation is defined, it follows that the number '0' can be output if and only if there is no j such that $A(j) = x$.

The above algorithm is of nondeterministic complexity $O(1)$. Note that since A is not ordered, every deterministic search algorithm is of complexity $\Omega(n)$. □

Example 11.2 [Sorting] Let $A(i)$, $1 \leq i \leq n$ be an unsorted set of positive integers. The nondeterministic algorithm NSORT(A, n) sorts the numbers into nondecreasing order and then outputs them in this order. An auxiliary array $B(1{:}n)$ is used for convenience. Line 1 initializes B to zero though any value different from all the $A(i)$ will do. In the loop of lines 2-6 each $A(i)$ is assigned to a position in B. Line 3 nondeterministically determines this position. Line 4 ascertains that $B(j)$ has not already been used. Thus, the order of the numbers in B is some permutation of the initial order in A. Lines 7 to 9 verify that B is sorted in nondecreasing order. A successful completion is achieved iff the numbers are output in nondecreasing order. Since there is always a set of choices at line 3 for such an output order, algorithm NSORT is a sorting algorithm. Its complexity is $O(n)$. Recall that all deterministic sorting algorithms must have a complexity $\Omega(n \log n)$. □

```
     procedure NSORT(A, n)
       //sort n positive integers//
       integer A(n), B(n), n, i, j
1      B ← 0  //initialize B to zero//
2      for i ← 1 to n do
3        j ← choice(1:n)
4        if B(j) ≠ 0 then failure endif
5        B(j) ← A(i)
6      repeat
7      for i ← 1 to n − 1 do    //verify order//
8        if B(i) > B(i + 1) then failure endif
9      repeat
10     print(B)
11     success
12   end NSORT
```

Algorithm 11.1 Nondeterministic sorting

A deterministic interpretation of a nondeterministic algorithm can be made by allowing unbounded parallelism in computation. Each time a choice is to be made, the algorithm makes several copies of itself. One copy is made for each of the possible choices. Thus, many copies executing at the same time. The first copy to reach a successful completion terminates all other computations. If a copy reaches a failure completion then only that copy of the algorithm terminates. Recall that the **success** and **failure** signals are equivalent to **stop** statements in deterministic algorithms. They may not be used in place of **return** statements. While this interpretation may enable one to better understand nondeterministic algorithms, it is important to remember that a nondeterministic machine does not make any copies of an algorithm every time a choice is to be made. Instead, it has the ability to select a "correct" element from the set of allowable choices (if such an element exists) every time a choice is to be made. A "correct" element is defined relative to a shortest sequence of choices that leads to a successful termination. In case there is no sequence of choices leading to a successful termination, we shall assume that the algorithm terminates in one unit of time with output "unsuccessful computation." Whenever successful termination is possible, a nondeterministic machine makes a sequence of choices which is a shortest sequence leading to a successful termination. Since, the machine we are defining is fictitious, it is not necessary for us to concern ourselves with how the machine can make a correct choice at each step.

It is possible to construct nondeterministic algorithms for which many different choice sequences lead to a successful completion. Procedure NSORT of Example 11.2 is one such algorithm. If the numbers $A(i)$ are not distinct then many different permutations will result in a sorted sequence. If NSORT were written to output the permutation used rather than the $A(i)$'s in sorted order then its output would not be uniquely defined. We shall concern ourselves only with those nondeterministic algorithms that generate a unique output. In particular we shall consider only *nondeterministic decision algorithms*. Such algorithms generate only a zero or one as their output. A binary decision is made. A successful completion is made iff the output is '1'. A '0' is output iff there is no sequence of choices leading to a successful completion. The output statement is implicit in the signals **success** and **failure**. No explicit output statements are permitted in a decision algorithm. Clearly, our earlier definition of a nondeterministic computation implies that the output from a decision algorithm is uniquely defined by the input parameters and the algorithm specification.

While the idea of a decision algorithm may appear very restrictive at this time, many optimization problems can be recast into decision problems with the property that the decision problem can be solved in polynomial time iff the corresponding optimization problem can. In other cases, we

can at least make the statement that if the decision problem cannot be solved in polynomial time then the optimization problem cannot either.

Example 11.3 [Max Clique] A maximal complete subgraph of a graph $G = (V, E)$ is a **clique**. The size of the clique is the number of vertices in it. The *max clique problem* is to determine the size of a largest clique in G. The corresponding decision problem is to determine if G has a clique of size at least k for some given k. Let DCLIQUE(G, k) be a deterministic decision algorithm for the clique decision problem. If the number of vertices in G is n, the size of a max clique in G can be found by making several applications of DCLIQUE. DCLIQUE is used once for each k, $k = n, n-1, n-2, \ldots$ until the output from DCLIQUE is 1. If the time complexity of DCLIQUE is $f(n)$ then the size of a max clique can be found in time $n*f(n)$. Also, if the size of a max clique can be determined in time $g(n)$ then the decision problem may be solved in time $g(n)$. Hence, the max clique problem can be solved in polynomial time iff the clique decision problem can be solved in polynomial time. □

Example 11.4 [0/1-Knapsack] The knapsack decision problem is to determine if there is a 0/1 assignment of values to x_i, $1 \leq i \leq n$ such that $\Sigma p_i x_i \geq R$ and $\Sigma w_i x_i \leq M$. R is a given number. The p_i's and w_i's are nonnegative numbers. Clearly, if the knapsack decision problem cannot be solved in deterministic polynomial time then the optimization problem cannot either. □

Before proceeding further, it is necessary to arrive at a uniform parameter, n, to measure complexity. We shall assume that n is the length of the input to the algorithm. We shall also assume that all inputs are integer. Rational inputs can be provided by specifying pairs of integers. Generally, the length of an input is measured assuming a binary representation. I.e., if the number 10 is to be input then, in binary it is represented as 1010. Its length is 4. In general, a positive integer k has a length of $\lfloor \log_2 k \rfloor + 1$ bits when represented in binary. The length of the binary representation of 0 is 1. The size or length, n, of the input to an algorithm is the sum of the lengths of the individual numbers being input. In case the input is given using a different representation (say radix r), then the length of a positive number k is $\lfloor \log_r k \rfloor + 1$. Thus, in decimal notation, $r = 10$ and the number 100 has a length $\log_{10} 100 + 1 = 3$ digits. Since $\log_r k = \log_2 k / \log_2 r$, the length of any input using radix $r (r > 1)$ representation is $c(r) \cdot n$ where n is the length using a binary representation and $c(r)$ is a number which is fixed for a given r.

When inputs are given using the radix $r = 1$, we say the input is in *unary form*. In unary form, the number 5 is input as 11111. Thus, the length of a positive integer k is k. It is important to observe that the length of a unary input is exponentially related to the length of the corresponding r-ary input for radix r, $r > 1$.

Example 11.5 [Max Clique] The input to the max clique decision problem may be provided as a sequence of edges and an integer k. Each edge in $E(G)$ is a pair of numbers (i, j). The size of the input for each edge (i, j) is $\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2$ if a binary representation is assumed. The input size of any instance is

$$n = \sum_{\substack{(i,j) \in E(G) \\ i<j}} (\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2) + \lfloor \log_2 k \rfloor + 1.$$

Note that if G has only one connected component then $n \geq |V|$. Thus, if this decision problem cannot be solved by an algorithm of complexity $p(n)$ for some polynomial $p(\ )$ then it cannot be solved by an algorithm of complexity $p(|V|)$. □

Example 11.6 [0/1 Knapsack] Assuming p_i, w_i, M and R are all integers, the input size for the knapsack decision problem is

$$m = \sum_{1 \leq i \leq n} (\lfloor \log_2 p_i \rfloor + \lfloor \log_2 w_i \rfloor) + \lfloor \log_2 M \rfloor + \lfloor \log_2 R \rfloor + 2n + 2.$$

Note that $m \geq n$. If the input is given in unary notation then the input size s is $\Sigma p_i + \Sigma w_i + M + R$. Note that the knapsack decision and optimization problems can be solved in time $p(s)$ for some polynomial $p(\ )$ (see the dynamic programming algorithm). However, there is no known algorithm with complexity $O(p(n))$ for some polynomial $p(\ )$. □

We are now ready to formally define the complexity of a nondeterministic algorithm.

Definition The *time required by a nondeterministic algorithm* performing on any given input is the minimum number of steps needed to reach a successful completion if there exists a sequence of choices leading to such a completion. In case successful completion is not possible then the time required is $O(1)$. A nondeterministic algorithm is of complexity $O(f(n))$ if for all inputs of size, n, $n \geq n_0$, that result in a successful completion the time required is at most $c \cdot f(n)$ for some constants c and n_0.

In the above definition we assume that each computation step is of a fixed cost. In word oriented computers this is guaranteed by the finiteness of each word. When each step is not of a fixed cost it is necessary to consider the cost of individual instructions. Thus, the additon of two m bit numbers takes $O(m)$ time, their multiplication takes $O(m^2)$ time (using classical multiplication) etc. To see the necessity of this consider procedure SUM (Algorithm 11.2). This is a deterministic algorithm for the sum of subsets decision problem. It uses an $M + 1$ bit word S. The i'th bit in S is zero iff no subset of the integers $A(j)$, $1 \leq j \leq n$ sums to i. Bit 0 of S is always 1 and the bits are numbered 0, 1, 2, ..., M right to left. The function SHIFT shifts the bits in S to the left by $A(i)$ bits. The total number of steps for this algorithm is only $O(n)$. However, each step moves $M + 1$ bits of data and would really take $O(M)$ time on a conventional computer. Assuming one unit of time is needed for each basic operation for a fixed word size, the true complexity is $O(nM)$ and not $O(n)$.

```
procedure SUM(A, n, M)
  integer A(n), S, n, M
  S ← 1  //S is an M + 1 bit word. Bit zero is 1//
  for i ← 1 to n do
    S ← S or SHIFT(S, A(i))
  repeat
  if Mth bit in S = 0 then print ('no subset sums to M')
                      else print ('a subset sums to M')
  endif
end SUM
```

Algorithm 11.2 Deterministic sum of subsets

The virtue of conceiving of nondeterministic algorithms is that often what would be very complex to write down deterministically is very easy to write nondeterministically. In fact, it is very easy to obtain polynomial time nondeterministic algorithms for many problems that can be deterministically solved by a systematic search of a solution space of exponential size.

Example 11.7 [Knapsack decision problem] Procedure DKP (Algorithm 11.3) is a nondeterministic polynomial time algorithm for the knapsack decision problem. Lines 1 to 3 assign 0/1 values to $X(i)$, $1 \leq i \leq n$. Line 4 checks to see if this assignment is feasible and if the resulting profit is at least R. A successful termination is possible iff the answer to the decision problem is yes. The time complexity is $O(n)$. If m is the input length using a binary representation, the time is $O(m)$. □

```
    procedure DKP(P, W, n, M, R, X)
      integer P(n), W(n), R, X(n), n, M, i
1     for i ← 1 to n do
2       X(i) ← choice (0, 1)
3     repeat
4     if Σ_{1≤i≤n} (W(i)*X(i)) > M or Σ_{1≤i≤n} (P(i)*X(i)) < R then failure
                                                              else success
5     endif
    end DKP
```

Algorithm 11.3 Nondeterministic Knapsack problem

Example 11.8 [Max Clique] Procedure DCK (Algorithm 11.4) is a nondeterministic algorithm for the clique decision problem. The algorithm begins by trying to form a set of k distinct vertices. Then it tests to see if these vertices form a complete subgraph. If G is given by its adjacency matrix and $|V| = n$, the input length m is $n^2 + \lfloor \log_2 k \rfloor + \lfloor \log_2 n \rfloor + 2$. Lines 2 to 6 can easily be implemented to run in nondeterministic time $O(n)$. The time for lines 7–10 is $O(k^2)$. Hence the overall nondeterministic time is $O(n + k^2) = O(n^2) = O(m)$. There is known polynomial time deterministic algorithm for this problem. □

```
    procedure DCK(G, n, k)
1     S ← φ  //S is an initially empty set//
2     for i ← 1 to k do  //select k distinct vertices//
3       t ← choice (1:n)
4       if t ∈ S then failure endif
5       S ← S ∪ t  //add t to set S//
6     repeat
      //at this point S contains k distinct vertex indices//
7     for all pairs (i, j) such that i ∈ S, j ∈ S and i ≠ j do
8       if (i, j) is not an edge of the graph
9         then failure endif
10    repeat
11    success
    end DCK
```

Algorithm 11.4 Nondeterministic clique

Example [Satisfiability] Let $x_1, x_2, \ldots,$ denote boolean variables (their value is either true or false). Let $\bar{x}_i$ denote the negation of x_i. A *literal* is either a variable or its negation. A formula in the propositional calculus is an expression that can be constructed using literals and the operations **and** and **or**. Examples of such formulas are $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$; $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$. $\vee$ denotes **or** and $\wedge$ denotes **and**. A formula is in *conjunctive normal form* (CNF) iff it is represented as $\wedge_{i=1}^{k} c_i$ where the c_i are clauses each represented as $\vee l_{ij}$. The l_{ij} are literals. It is in *disjunctive normal form* (DNF) iff it is represented as $\vee_{i=1}^{k} c_i$ and each clause c_i is represented as $\wedge l_{ij}$. Thus $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ is in DNF while $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$ is in CNF. The *satisfiability* problem is to determine if a formula is true for any assignment of truth values to the variables. *CNF-satisfiability* is the satisfiability problem for CNF formulas.

It is easy to obtain a polynomial time nondeterministic algorithm that terminates successfully if and only if a given propositional formula $E(x_1, \ldots, x_n)$ is satisfiable. Such an algorithm could proceed by simply choosing (nondeterministically) one of the 2^n possible assignments of truth values to $(x_1, \ldots, x_n)$ and verifying that $E(x_1, \ldots, x_n)$ is true for that assignment.

Procedure EVAL (Algorithm 11.5) does this. The nondeterministic time required by the algorithm is $O(n)$ to choose the value of $(x_1, \ldots, x_n)$ plus the time needed to deterministically evaluate E for that assignment. This time is proportional to the length of E. □

```
procedure EVAL(E, n)
  //Determine if the propositional formula E is satisfiable. The variables//
  //are x_i, 1 ≤ i ≤ n//
  boolean x(n)
  for i ← 1 to n do   //choose a truth value assignment//
    x_i ← choice (true, false)
  repeat
  if E(x_1, ..., x_n) is true then success   //satisfiable//
                      else failure
  endif
end EVAL
```

Algorithm 11.5 Nondeterministic satisfiability

The Classes NP-hard and NP-complete

In measuring the complexity of an algorithm we shall use the input length as the parameter. An algorithm A is of *polynomial complexity* if there exists a polynomial $p(\ )$ such that the computing time of A is $O(p(n))$ for every input of size n.

Definition P is the set of all decision problems solvable by a deterministic algorithm in polynomial time. NP is the set of all decision problems solvable by a nondeterministic algorithm in polynomial time.

Since deterministic algorithms are just a special case of nondeterministic ones, we can conclude that $P \subseteq NP$. What we do not know, and what has become perhaps the most famous unsolved problem in computer science is whether $P = NP$ or $P \neq NP$.

Is it possible that for all of the problems in NP there exist polynomial time deterministic algorithms which have remained undiscovered? This seems unlikely, at least because of the tremendous effort which has already been expended by so many people on these problems. Nevertheless, a proof that $P \neq NP$ is just as elusive and seems to require as yet undiscovered techniques. But as with many famous unsolved problems, they serve to generate other useful results, and the $P \stackrel{?}{=} NP$ question is no exception.

In considering this problem S. Cook formulated the following question: Is there any single problem in NP such that if we showed it to be in P, then that would imply that $P = NP$. Cook answered his own question in the affirmative with the following theorem.

Theorem 11.1 (Cook) Satisfiability is in P if and only if $P = NP$.

Proof: See Section 11.2 □

We are now ready to define the NP-hard and NP-complete classes of problems. First we define the notion of reducibility.

Definition Let L_1 and L_2 be problems. L_1 *reduces to* L_2 (also written $L_1 \propto L_2$) if and only if there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time.

This definition implies that if we have a polynomial time algorithm for L_2 then we can solve L_1 in polynomial time. One may readily verify that $\propto$ is a transitive relation (i.e. if $L_1 \propto L_2$ and $L_2 \propto L_3$ then $L_1 \propto L_3$).

Definition A problem L is *NP-hard* if and only if satisfiability reduces

to L. (satisfiability $\propto L$). A problem L is *NP-complete* if and only if L is *NP-hard* and $L \in NP$.

It is easy to see that there are *NP*-hard problems that are not *NP*-complete. Only a decision problem can be NP-complete. However, an optimization problem may be NP-hard. Furthermore if L_1 is a decision problem and L_2 an optimization problem, it is quite possible that $L_1 \propto L_2$. One may trivially show that the knapsack decision problem reduces to the knapsack optimization problem. For the clique problem one may easily show that the clique decision problem reduces to the clique optimization problem. In fact, we can also show that these optimization problems reduce to their corresponding decision problems (see exercises). Yet, optimization problems cannot be NP-complete while decision problems can. There also exist NP-hard decision problems that are not NP-complete.

Example 11.10 As an extreme example of an NP-hard decision problem that is not NP-complete consider the halting problem for deterministic algorithms. The *halting problem* is to determine for an arbitrary deterministic algorithm A and an input I whether algorithm A with input I ever terminates (or enters an infinite loop). It is well known that this problem is undecidable. Hence, there exists no algorithm (of any complexity) to solve this problem. So, it clearly cannot be in NP. To show satisfiability $\propto$ halting problem simply construct an algorithm A whose input is a propositional formula X. If X has n variables then A tries out all 2^n possible truth assignments and verifies if X is satisfiable. If it is then A stops. If X is not satisfiable then A enters an infinite loop. Hence, A halts on input X iff X is satisfiable. If we had a polynomial time algorithm for the halting problem then we could solve the satisfiability problem in polynomial time using A and X as input to the algorithm for the halting problem. Hence, the halting problem is an NP-hard problem which is not in NP. □

Definition Two problems L_1 and L_2 are said to be *polynomially equivalent* iff $L_1 \propto L_2$ and $L_2 \propto L_1$.

In order to show that a problem, L_2 is NP-hard it is adequate to show $L_1 \propto L_2$ where L_1 is some problem already known to be NP-hard. Since $\propto$ is a transitive relation, it follows that if satisfiability $\propto L_1$ and $L_1 \propto L_2$ then satisfiability $\propto L_2$. *To show an NP-hard decision problem NP-complete we have just to exhibit a polynomial time nondeterministic algorithm for it.* Later sections will show many problems to be NP-hard. While we shall restrict ourselves to decision problems, it should be clear that the corresponding optimization problems are also NP-hard. The NP-completeness proofs will be left as exercises (for those problems that are NP-complete).

11.2 COOK'S THEOREM

Cook's theorem (Theorem 11.1) states that satisfiability is in P iff P = NP. We shall now prove this important theorem. We have already seen that satisfiability is in NP (Example 11.9). Hence, if P = NP then satisfiability is in P. It remains to be shown that if satisfiability is in P then P = NP. In order to prove this latter statement, we shall show how to obtain from any polynomial time nondeterministic decision algorithm A and input I a formula $Q(A, I)$ such that Q is satisfiable iff A has a successful termination with input I. If the length of I is n and the time complexity of A is $p(n)$ for some polynomial $p(\ )$ then the length of Q will be $O(p^3(n) \log n) = O(p^4(n))$. The time needed to construct Q will also be $O(p^3(n) \log n)$. A deterministic algorithm Z to determine the outcome of A on any input I may be easily obtained. Z simply computes Q and then uses a deterministic algorithm for the satisfiability problem to determine whether or not Q is satisfiable. If $O(q(m))$ is the time needed to determine if a formula of length m is satisfiable then the complexity of Z is $O(p^3(n) \log n + q(p^3(n) \log n))$. If satisfiability is in P then $q(m)$ is a polynomial function of m and the complexity of Z becomes $O(r(n))$ for some polynomial $r(\ )$. Hence, if satisfiability is in P then for every nondeterministic algorithm A in NP we can obtain a deterministic Z in P. So, the above construction will show that if satisfiability is in P then P = NP.

Before going into the construction of Q from A and I, we shall make some simplifying assumptions on our nondeterministic machine model and on the form of A. These assumptions will not in any way alter the class of decision problems in NP or P. The simplifying assumptions are:

i) The machine on which A is to be executed is word oriented. Each word is w bits long. Multiplication, addition, subtraction etc. between numbers one word long take one unit of time. In case numbers are longer than a word then the corresponding operations take at least as many units as the number of words making up the longest number.

ii) A *simple expression* is an expression that contains at most one operator and all operands are simple variables (i.e., no array variables are used). Some sample simple expressions are $-B$, $B + C$, D or E, F.