THEOREM 7.14

$PATH \in \mathrm{P}$.

PROOF IDEA We prove this theorem by presenting a polynomial time algorithm that decides *PATH*. Before describing that algorithm, let's observe that a brute-force algorithm for this problem isn't fast enough.

A brute-force algorithm for *PATH* proceeds by examining all potential paths in G and determining whether any is a directed path from s to t. A potential path is a sequence of nodes in G having a length of at most m, where m is the number of nodes in G. (If any directed path exists from s to t, one having a length of at most m exists because repeating a node never is necessary.) But the number of such potential paths is roughly m^m, which is exponential in the number of nodes in G. Therefore, this brute-force algorithm uses exponential time.

To get a polynomial time algorithm for *PATH*, we must do something that avoids brute force. One way is to use a graph-searching method such as breadth-first search. Here, we successively mark all nodes in G that are reachable from s by directed paths of length 1, then 2, then 3, through m. Bounding the running time of this strategy by a polynomial is easy.

PROOF A polynomial time algorithm M for *PATH* operates as follows.

$M =$ "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

Now we analyze this algorithm to show that it runs in polynomial time. Obviously, stages 1 and 4 are executed only once. Stage 3 runs at most m times because each time except the last it marks an additional node in G. Thus, the total number of stages used is at most $1 + 1 + m$, giving a polynomial in the size of G.

Stages 1 and 4 of M are easily implemented in polynomial time on any reasonable deterministic model. Stage 3 involves a scan of the input and a test of whether certain nodes are marked, which also is easily implemented in polynomial time. Hence M is a polynomial time algorithm for *PATH*.

Let's turn to another example of a polynomial time algorithm. Say that two numbers are ***relatively prime*** if 1 is the largest integer that evenly divides them both. For example, 10 and 21 are relatively prime, even though neither of them is a prime number by itself, whereas 10 and 22 are not relatively prime because

both are divisible by 2. Let *RELPRIME* be the problem of testing whether two numbers are relatively prime. Thus

$$\mathit{RELPRIME} = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime}\}.$$

THEOREM 7.15

$\mathit{RELPRIME} \in \mathrm{P}$.

PROOF IDEA One algorithm that solves this problem searches through all possible divisors of both numbers and accepts if none are greater than 1. However, the magnitude of a number represented in binary, or in any other base k notation for $k \geq 2$, is exponential in the length of its representation. Therefore, this brute-force algorithm searches through an exponential number of potential divisors and has an exponential running time.

Instead, we solve this problem with an ancient numerical procedure, called the ***Euclidean algorithm***, for computing the greatest common divisor. The ***greatest common divisor*** of natural numbers x and y, written $\gcd(x, y)$, is the largest integer that evenly divides both x and y. For example, $\gcd(18, 24) = 6$. Obviously, x and y are relatively prime iff $\gcd(x, y) = 1$. We describe the Euclidean algorithm as algorithm E in the proof. It uses the mod function, where $x \bmod y$ is the remainder after the integer division of x by y.

PROOF The Euclidean algorithm E is as follows.

$E =$ "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

1. Repeat until $y = 0$:
2. Assign $x \leftarrow x \bmod y$.
3. Exchange x and y.
4. Output x."

Algorithm R solves *RELPRIME*, using E as a subroutine.

$R =$ "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

1. Run E on $\langle x, y\rangle$.
2. If the result is 1, *accept*. Otherwise, *reject*."

Clearly, if E runs correctly in polynomial time, so does R and hence we only need to analyze E for time and correctness. The correctness of this algorithm is well known so we won't discuss it further here.

To analyze the time complexity of E, we first show that every execution of stage 2 (except possibly the first) cuts the value of x by at least half. After stage 2 is executed, $x < y$ because of the nature of the mod function. After stage 3, $x > y$ because the two have been exchanged. Thus, when stage 2 is subsequently

executed, $x > y$. If $x/2 \geq y$, then $x \bmod y < y \leq x/2$ and x drops by at least half. If $x/2 < y$, then $x \bmod y = x - y < x/2$ and x drops by at least half.

The values of x and y are exchanged every time stage 3 is executed, so each of the original values of x and y are reduced by at least half every other time through the loop. Thus, the maximum number of times that stages 2 and 3 are executed is the lesser of $2 \log_2 x$ and $2 \log_2 y$. These logarithms are proportional to the lengths of the representations, giving the number of stages executed as $O(n)$. Each stage of E uses only polynomial time, so the total running time is polynomial.

The final example of a polynomial time algorithm shows that every context-free language is decidable in polynomial time.

THEOREM 7.16

Every context-free language is a member of P.

PROOF IDEA In Theorem 4.9, we proved that every CFL is decidable. To do so, we gave an algorithm for each CFL that decides it. If that algorithm runs in polynomial time, the current theorem follows as a corollary. Let's recall that algorithm and find out whether it runs quickly enough.

Let L be a CFL generated by CFG G that is in Chomsky normal form. From Problem 2.26, any derivation of a string w has $2n - 1$ steps, where n is the length of w because G is in Chomsky normal form. The decider for L works by trying all possible derivations with $2n - 1$ steps when its input is a string of length n. If any of these is a derivation of w, the decider accepts; if not, it rejects.

A quick analysis of this algorithm shows that it doesn't run in polynomial time. The number of derivations with k steps may be exponential in k, so this algorithm may require exponential time.

To get a polynomial time algorithm, we introduce a powerful technique called ***dynamic programming***. This technique uses the accumulation of information about smaller subproblems to solve larger problems. We record the solution to any subproblem so that we need to solve it only once. We do so by making a table of all subproblems and entering their solutions systematically as we find them.

In this case, we consider the subproblems of determining whether each variable in G generates each substring of w. The algorithm enters the solution to this subproblem in an $n \times n$ table. For $i \leq j$, the (i, j)th entry of the table contains the collection of variables that generate the substring $w_i w_{i+1} \cdots w_j$. For $i > j$, the table entries are unused.

The algorithm fills in the table entries for each substring of w. First it fills in the entries for the substrings of length 1, then those of length 2, and so on.