no known polynomial time algorithms are computationally related. In fact, we establish two classes of problems. These are given the names $\mathcal{NP}$-hard and $\mathcal{NP}$-complete. A problem that is $\mathcal{NP}$-complete has the property that it can be solved in polynomial time if and only if all other $\mathcal{NP}$-complete problems can also be solved in polynomial time. If an $\mathcal{NP}$-hard problem can be solved in polynomial time, then all $\mathcal{NP}$-complete problems can be solved in polynomial time. All $\mathcal{NP}$-complete problems are $\mathcal{NP}$-hard, but some $\mathcal{NP}$-hard problems are not known to be $\mathcal{NP}$-complete.

Although one can define many distinct problem classes having the properties stated above for the $\mathcal{NP}$-hard and $\mathcal{NP}$-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 $\mathcal{NP}$-complete or $\mathcal{NP}$-hard problem is polynomially solvable.

We see that the class of $\mathcal{NP}$-hard problems (and the subclass of $\mathcal{NP}$-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.

11.1.1 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 algorithms 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 outcomes are not uniquely defined but are limited to specified sets 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 three new functions:

1. Choice(S) arbitrarily chooses one of the elements of set S.

2. Failure() signals an unsuccessful completion.

3. Success() signals a successful completion.

The assignment statement $x :=$ 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 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 unsuccessfully 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*. Although nondeterministic machines (as defined here) do not exist in practice, we see that they 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 Algorithm 11.1.

```
$j :=$ Choice$(1, n)$;
**if** $A[j] = x$ **then** {**write** $(j)$; Success();}
**write** $(0)$; Failure();
```

Algorithm 11.1 Nondeterministic search

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 $A[j] = x$. Algorithm 11.1 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 array of positive integers. The nondeterministic algorithm NSort(A, n) (Algorithm 11.2) sorts the numbers into nondecreasing order and then outputs them in this order. An auxiliary array $B[1:n]$ is used for convenience. Line 4 initializes B to zero though any value different from all the $A[i]$ will do. In the **for** loop of lines 5 to 10, each $A[i]$ is assigned to a position in B. Line 7 nondeterministically determines this position. Line 8 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. The **for** loop of lines 11 and 12 verifies that B is sorted in nondecreasing order. A successful completion is achieved if and only if the numbers are output in nondecreasing order. Since there is always a set of choices at line 7 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 $\Omega(n \log n)$. □

```
Algorithm NSort(A, n)
// Sort n positive integers.
{
    for i := 1 to n do B[i] := 0; // Initialize B[ ].
    for i := 1 to n do
    {
        j := Choice(1, n);
        if B[j] ≠ 0 then Failure();
        B[j] := A[i];
    }
    for i := 1 to n − 1 do  // Verify order.
        if B[i] > B[i + 1] then Failure();
    write (B[1 : n]);
    Success();
}
```

Algorithm 11.2 Nondeterministic sorting

A deterministic interpretation of a nondeterministic algorithm can be made by allowing unbounded parallelism in computation. In theory, 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 are 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. Although 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 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 that 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.

Definition 11.1 Any problem for which the answer is either zero or one is called a *decision problem*. An algorithm for a decision problem is termed