# COMPUTER ALGORITHMS

Sartaj Sahni *University of Minnesota*

## GLOSSARY

**Algorithm:** Sequence of well-defined instructions the execution of which results in the solution of a specific problem. The instructions are unambiguous, and each can be performed in a finite amount of time. Furthermore, the execution of all the instructions together takes only a finite amount of time.

**Approximation algorithm:** Algorithm that is guaranteed to produce solutions whose value is within some prespecified amount of the value of an optimal solution.

**Asymptotic analysis:** Analysis of the performance of an algorithm for large problem instances. Typically, the time and space requirements are analyzed and provided as a function of parameters that reflect properties of the problem instance to be solved. Asymptotic notation (e.g., big "oh," theta, omega) is used.

**Deterministic algorithm:** Algorithm in which the outcome of each step is well defined and determined by the values of the variables (if any) involved in the step. For example, the value of $x + y$ is determined by the values of $x$ and $y$.

**Heuristic:** Rule of thumb employed in an algorithm to improve its performance (time and space requirements or quality of solution produced). This rule may be very effective in certain instances and ineffective in others.

**Lower bound:** Defined with respect to a problem. A lower bound on the resources (time or space) needed to solve a specified problem has the property that the problem cannot be solved by any algorithm that uses less resource than the lower bound.

**Nondeterministic algorithm:** Algorithm that may contain some steps whose outcome is determined by selecting from a set of permissible outcomes. There are no rules determining how the selection is to be made. Rather, such an algorithm terminates in one of two modes: success and failure. It is required that, whenever possible, the selection of the outcomes of individual steps be done in such a way that the algorithm terminates successfully.

**NP-Complete problem:** Decision problem (one for which the solution is "yes" or "no") that has the following property. The decision problem can be solved in polynomial deterministic time iff all decision problems that can be solved in nondeterministic polynomial time are also solvable in deterministic polynomial time.

**NP-Hard problem:** Problem for which the following is true. If this problem can be solved in polynomial deterministic time, then all decision problems that can be solved in nondeterministic polynomial time are also solvable in deterministic polynomial time.

**Performance:** Amount of resources (i.e., amount of computer time and memory) required by an algorithm. If the algorithm does not guarantee optimal solutions, the term *performance* is also used to include some measure of the quality of the solutions produced.

**Probabilistically good algorithm:** Algorithm that does not guarantee optimal solutions but generally does provide them.

Copyright © 1987 by Academic Press, Inc.
All rights of reproduction in any form reserved.

**Simulated annealing:** Combinatorial optimization technique adapted from statistical mechanics. The technique attempts to find solutions that have value close to optimal. it does so by simulating the physical process of annealing a metal.

**Stepwise refinement:** Program development method in which the final computer program is arrived at in a sequence of steps. The first step begins close to the problem specification. Each step is a refinement of the preceding one and gets one closer to the final program. This technique simplifies both the programming task and the task of proving the final program correct.

**Usually good algorithm:** Algorithm that generally provides optimal solutions using a small amount of computing resources. At other times, the resources required may be prohibitively large.

In order to get a computer to solve a problem, it is necessary to provide it with a sequence of instructions that if followed faithfully will result in the desired solution. This sequence of instructions is called a computer algorithm. When a computer algorithm is specified in a language the computer "understands" (i.e., a programming language), it is called a program. The topic of computer algorithms deals with methods of developing algorithms as well as methods of analyzing algorithms to determine the amount of computer resources (time and memory) required by them to solve a problem and methods of deriving lower bounds on the resources required by any algorithm to solve a specific problem. Finally, for certain problems that are difficult to solve (e.g., when the computer resources required are impractically large), heuristic methods are used. [*See* COMPUTER LOGIC; PROGRAMMING LANGUAGES.]

## I. Computers and Algorithms

We begin by discussing the two words—*computer* and *algorithm*—that comprise the title of this article. The word *computer* is defined in *Webster's Third New International Dictionary* as "one that computes." By this definition, almost every human is a computer. Certainly each of us has at one time or other computed the number of days before school ends. More serious computing has been done by humans for thousands of years. In fact, the Babylonians used formulas to compute the areas of surfaces and the volumes of solids.

Generally, however, the word *computer* refers to a nonhuman device that computes. Early devices in this category include the sun dial and the Chinese abacus. Contemporary computing devices include microcomputers, such as the Apple® and the IBM-PC®, and awesome supercomputers, such as the CRAY-XMP® and NASA's MPP®. These contemporary computing devices have had a very significant impact on our lives—far more significant than that of any other computing device created by humankind. Why is this so? [*See* COMPUTER ARCHITECTURE.]

There are essentially two reasons. The first is that the modern computer is significantly faster than any of its predecessors. Some home computers are capable of performing several hundred thousand computations per second, and the fastest supercomputers exceed a billion computations per second. This awesome speed, in itself, is not enough. Equally important is the capacity of modern computers to store programs and data. (A *program* is simply a sequence of instructions to the computer.)

To understand the significance of the latter reason, consider the following problem:

Mary intends to open a bank account with an initial deposit of \$100. She intends to deposit an additional \$100 into this account on the first day of each of the next 19 months for a total of 20 deposits (including the initial deposit). The account pays interest at a rate of 5% per annum compounded monthly. Her initial deposit is also on the first of the month. Mary would like to know what the balance in her account will be at the end of each of the 20 months in which she will be making a deposit.

In order to solve this problem, we need to know how much interest is earned each month. Since the annual interest rate is 5%, the monthly interest rate is $\frac{5}{12}\%$. Consequently, the balance at the end of a month is

$$\begin{aligned}&(\text{initial balance} + \text{interest})\\ &\quad = (\text{initial balance}) * (1 + \tfrac{5}{1200})\\ &\quad = \tfrac{241}{240}\,(\text{initial balance})\end{aligned}$$

Having performed this analysis, we can proceed to compute the balance at the end of each month using the following steps:

1. Let *balance* denote the current balance. The starting balance is \$100, so set *balance* = 100.
2. The balance at the end of the month is $\frac{241}{240} * balance$. Update *balance*.
3. If 20 months have not elapsed, then add 100 to *balance* to reflect the deposit for the next month. Go to step 2. Otherwise, we are done.

Suppose that we have to compute the monthly balances using a computing device that cannot store the computational steps and associated data. A nonprogrammable calculator is one such device. The above steps will translate into the following process:

1. Turn the calculator on.
2. Enter the initial balance as the number 100.
3. Multiply by 241 and then divide by 240.
4. Note the result down as a monthly balance.
5. If the number of monthly balances noted down is 20, then stop.
6. Otherwise, add 100 to the previous result.
7. Go to step 3.

If we tried this process on any electronic calculator, we would notice that the total time spent is not determined by the speed of the calculator. Rather, it is determined by how fast we can enter the required numbers and operators (add, multiply, etc.) and how fast we can copy the monthly balances. Even if the calculator could perform a billion computations per second, we would not be able to solve the above problem any faster.

When a stored-program computing device is used, the above instructions need be entered into the computer only once. The computer can then sequence through these instructions at its own speed. Since the instructions are entered just once (rather than 20 times), we get almost a 20-fold speedup in the computation. If the balance for 1000 months is required, the speedup is by a factor of almost 1000. We have achieved this speedup without making our computing device any faster. We have simply cut down on the input work required by the slow human!

Instruction sequences are provided to a computer by means of a programming language. There are more than a thousand programming languages in use today. Some of the popular ones are Basic, COBOL, FORTRAN, and Pascal. The seven-step computational process stated above translates into the Basic program shown in Program 1. In Pascal, this takes the form shown in Program 2.

PROGRAM 1. Basic Program for Mary's Problem

```
10 balance = 100
20 month = 1
30 balance = 241 * balance/240
40 print month, "$"; balance
50 if month = 20 then stop
60 month = month + 1
70 balance = balance + 100
80 goto 30
```

Apart from the fact that these two programs have been written in different languages, they represent different programming styles. The Pascal program has been written in such a way as to permit one to make changes with ease. The number of months, interest rate, initial balance, and monthly additions are more easily changed in the Pascal program.

Now let us turn our attention to the concept of an algorithm. *Webster's Third New International Dictionary* defines an algorithm as "any special method for solving a certain kind of problem." As in the case of the word *computer,* we need a more precise definition of the word *algorithm.* We shall arrive at this definition in two steps. First, we define a *computational procedure*.

A computational procedure consists of a finite number of steps, each of which is definite and effective. By definite, we mean that the outcome of each is well defined. The step "Set $x$ to 1/0" is not definite, as 1/0 is not a well-defined quantity. Each of the steps in our seven-step calculator example and in the corresponding Basic and Pascal programs is well defined. By effective, we mean that the step can be completed in a finite amount of time using a finite amount of computational resource. It is easy to see that each of the steps in our calculator, Basic, and Pascal examples is effective. Hence, the three forms of our solution to Mary's problem are all computational procedures.

A computational procedure that terminates on every input after executing a finite number of steps is called an *algorithm.* In obtaining the execution step count, each repetition counts as an additional step. So a single step executed seven times counts as much as seven different instructions executed once each. The computational procedures we have seen so far are all algorithms. Each terminates after executing a finite number of steps. The following sequence of Ba-

PROGRAM 2. Pascal Program for Mary's Problem

```
program account(input, output);
{compute the account balance at the end of each month}
const InitialBalance = 100;
        MonthlyDeposit = 100; {additional deposit per month}
        TotalMonths = 20;
        AnnualInterestRate = 5; {percent rate}
var balance, interest, MonthlyRate: real; month: integer;
begin
      MonthlyRate :=AnnualInterestRate/1200; {rate per $}
      balance :=InitialBalance;
      writeln('      Month       Balance');
      for month := 1 to TotalMonths do
      begin
         interest :=balance*MonthlyRate;
         balance :=balance+interest;
         writeln(month:10,'       ', balance:10:2);
         balance :=balance+MonthlyDeposit;
      end;
      writeln;
      writeln('Balance is balance at end of month');
end.
```

sic steps is a computational procedure that is not an algorithm:

```
10 x = 1
20 goto 10
```

Generally, we want our computational procedures to be algorithms. However, in some real situations (such as in a nuclear reactor monitoring environment) the procedure should never terminate.

Algorithms are stated in a variety of languages. These range from natural languages (such as English) to a combination of natural and programming languages to pure programming language. In arriving at a computer program, one generally starts with an English statement (or something very close to this) of the algorithm and then refines this into a computer program. Between the initial and final statements of the algorithms we may have several statements that employ a combination of English and programming language constructs. This process of developing a computer algorithm (or program) is called *stepwise refinement*. We shall see an example of this in the next section.

## II. Algorithm Design

There are several design techniques available to the designer of a computer algorithm. Some of the most successful techniques are the following:

1. Divide and conquer
2. Greedy method
3. Dynamic programming
4. Branch and bound
5. Backtracking

While we do not have the space here to elaborate each of these, we shall develop two algorithms using the divide-and-conquer technique. The essential idea in divide-and-conquer is to decompose a large problem instance into several smaller instances, solve the smaller instances, and combine the results (if necessary) to obtain the solution of the original problem instance.

The problem we shall investigate is that of sorting a sequence $x[1], x[2], \ldots, x[n]$ of $n$, $n \geq 0$, numbers; $n$ is the size of the instance. We wish to rearrange these numbers so that they are in nondecreasing order (i.e., $x[1] \leq x[2] \leq \cdots \leq x[n]$). For example, if $n = 5$, and $(x[1], \ldots, x[5]) = (10, 18, 8, 12, 9)$, then after the sort, the numbers are in the order (8, 9, 10, 12, 18).

Even before we attempt an algorithm to solve this problem, we can write down an English version of the solution, as in Program 3. The correctness of this version of the algorithm is immediate.

PROGRAM 3. First Version of Sort Algorithm

```
procedure sort;
   sort x[i], 1 ≤ i ≤ n into nondecreasing order;
end; {of sort}
```

Using the divide-and-conquer methodology, we first decompose the sort instance into several smaller instances. At this point, we must determine the size and number of these smaller instances. Some possibilities are the following:

(a) One of size $n - 1$ and another of size 1
(b) Two of approximately equal size
(c) $k$ of size approximately $n/k$ each, for some integer $k$, $k > 2$

We shall pursue the first two possibilities. In each of these, we have two smaller instances created. Using the first possibility, we can decompose the instance (10, 18, 8, 12, 9) into any of the following pairs of instances:

(a) (10, 18, 8, 12) (9)
(b) (10) (18, 8, 12, 9)
(c) (10, 18, 8, 9) (12)
(d) (10, 18, 12, 9) (8)

and so on. Suppose we choose the first option. Having decomposed the initial instance into two, we must sort the two instances and then combine the two sorted sequences into one. When (10, 18, 8, 12) is sorted, the result is (8, 10, 12, 18). Since the second sequence is of size 1, it is already in sorted order. To combine the two sequences, the number 9 must be inserted into the first sequence to get the desired five-number sorted sequence.

The preceding discussion raises two questions. How is the four-number sequence sorted? How is the one-number sequence inserted into the sorted four-number sequence? The answer to the first is that this, too, can be sorted using the divide-and-conquer approach. That is, we decompose it into two sequences: one of size 3 and the other of size 1. We then sort the sequence of size 3 and then insert the one-element sequence. To sort the three-element sequence, we decompose it into two sequences of size 2 and 1, respectively. To sort the sequence of size 2, we decompose it into two of size 1 each. At this point, we need merely insert one into the other.

Before attempting to answer the second question, we refine Program 3, incorporating the above discussion. The result is Program 4. Program 4 is a recursive statement of the sort algorithm being developed. In a recursive statement of an algorithm, the solution for an instance of size $n$ is defined in terms of solutions for instances of smaller size. In Program 4, the sorting of $n$ items, for $n > 1$, is defined in terms of the sorting of $n - 1$ items. This just means that to sort $n$ items using procedure *sort,* we must first use this procedure to sort $n - 1$ items. This in turn means that the procedure must first be used to sort $n - 2$ items, and so on. This use of recursion generally poses no problems, as most contemporary programming languages support recursive programs.

PROGRAM 4. Refinement of Program 3

```
procedure sort(n);
{sort n numbers into nondecreasing order}
if n > 1 then begin
              sort(n - 1); {sort first sequence}
              insert(n - 1, x[n]);
            end;
end; {of sort}
```

To refine Program 4, we must determine how an insert is performed. Let us consider an example. Consider the insertion of 9 into (8, 10, 12, 18). We begin by removing 9 from position 5 of the sequence and then comparing 9 and 18. Since 18 is larger, it must be to the right of 9. So 18 is moved to position 5. The resulting sequence is as follows ("–" denotes an empty position in the sequence):

8 10 12 – 18

Next, 9 is compared with 12, and 12 is moved to position 4. This results in the following sequence:

8 10 – 12 18

Then 9 is compared with 10, and 10 moved to position 3. At this time, we have the sequence

8 – 10 12 18

Finally, 9 is compared with 8. Since 9 is not smaller than 8, it is inserted into position 2. This results in the sequence (8, 9, 10, 12, 18).

With this discussion, we can refine Program 4 to get Program 5. Program 5 is then easily refined to get the Pascal-like code of Program 6. This code uses position 0 of the sequence to handle insertions into position 1. The recursion in this procedure can be eliminated to get Program 7.

The algorithm we have just developed for sorting is called *insertion sort.* This algorithm was obtained using the stepwise refinement process beginning with Program 3. As a result of using this process, we have confidence in the correctness of the resulting algorithm. Formal correctness proofs can be obtained using mathematical induction or other program verification methods.

PROGRAM 5. Refinement of Program 4

```
procedure sort(n);
{sort n numbers into nondecreasing order}
if n > 1 then begin
              sort(n - 1); {sort first sequence}
              assign t the value x[n];
              compare t with the xs beginning at x[n - 1];
              move the xs up until the correct place for t is found;
              insert t into this place;
           end;
end; {of sort}
```

PROGRAM 6. Refinement of Program 5

```
procedure sort(n);
{sort n numbers into nondecreasing order}
if n > 1 then begin
              sort(n - 1); {sort first sequence}
              assign t and x[0] the value x[n];
              assign i the value n - 1;
              while t < x[i] do {find correct place for t}
              begin
                 move x[i] to x[i + 1];
                 reduce i by 1;
              end;
              put t into x[i + 1];
           end;
end; {of sort}
```

Program 7 is quite close to being a Pascal program. One last refinement gives us a correct Pascal procedure to sort. This is given in Program 8. This procedure assumes that the numbers to be sorted are of type **integer**. In case numbers of a different type are to be sorted, the type declaration of $t$ should be changed. Another improvement of Program 8 can be made. This involves taking the statement $x[0] := t$ out of the **for** loop and initializing $x[0]$ to a very small number before the loop. The Basic code segment obtained by refining Program 7 is given in Program 9.

PROGRAM 7. Refinement of Program 6

```
procedure sort(n);
{sort n numbers into nondecreasing order}
for j := 2 to n do
begin {insert x[j] into x[1:j - 1]}
   assign t and x[0] the value x[j];
   assign i the value j - 1;
   while t < x[i] do {find correct place for t}
   begin
      move x[i] to x[i + 1];
      reduce i by 1;
   end;
   put t into x[i + 1];
end; {of for}
end; {of sort}
```

Let us consider the route our development process would have taken if we had decided to decompose sort instances into two smaller instances of roughly equal size. Let us further suppose that the left half of the sequence is one of the instances created and that the right half is the other. For our example, we get the instances (10, 18) and (8, 12, 9). These are sorted independently to get the sequences (10, 18) and (8, 9, 12). Next, the two sorted sequences are combined to get the sequence (8, 9, 10, 12, 18). This combination process is called *merging*. The resulting sort algorithm is called *merge sort*.

Program 10 is the refinement of Program 3 that results for merge sort. We shall not refine this further here. The reader will find complete programs for this algorithm in several of the references cited later. In Program 10, the notation $\lfloor x \rfloor$ is used. This is called the *floor* of $x$ and denotes the largest integer less than or equal to $x$. For example, $\lfloor 2.5 \rfloor = 2$, $\lfloor -6.3 \rfloor = -7$, $\lfloor 5/3 \rfloor = 1$, and

PROGRAM 8. Pascal Version of Program 7

```
procedure sort(n: integer);
{sort n numbers of type integer into nondecreasing order}
var t, i, j: integer;
begin
  for j := 2 to n do
  begin {insert x[j] into x[1:j - 1]}
    t := x[j]; x[0] := t; i := j - 1;
    while t < x[i] do {find correct place for t}
    begin
      x[i + 1] := x[i];
      i := i - 1;
    end;
    x[i + 1] := t;
  end; {of for}
end; {of sort}
```

PROGRAM 9. IBM-PC Basic version of Program 7

```
rem Sort n numbers x(1 ... n) into nondecreasing order
for j = 2 to n
rem Insert x(j) into x(1 ... j - 1)
  t = x(j): x(0) = t: i = j - 1

  rem Find correct place for t
  while t < x(i)
    x(i + 1) = t: i = i - 1
  wend
  x(i + 1) = t
next
```

PROGRAM 10. Merge Sort

```
procedure MergeSort(X, n);
{sort the n numbers in X}
if n > 1 then
begin
  Divide X into two sequences A and B such that A contains ⌊n/2⌋ numbers and
  B contains the remaining numbers;
  MergeSort(A, ⌊n/2⌋);
  MergeSort(B, n− ⌊n/2⌋);
  merge(A, B);
end; {of if}
end; {of MergeSort}
```

$\lfloor n/2 \rfloor$ denotes the largest integer less than or equal to $n/2$.

## III. Performance Analysis and Measurement

In the preceding section, we developed two algorithms for sorting. Which of these should we use? The answer to this depends on the relative performance of the two algorithms.

The performance of an algorithm is measured in terms of the space and time needed by the algorithm to complete its task. Let us concentrate on time here. In order to answer the question "How much time does insertion sort take?" we must ask ourselves the following:

1. What is the instance size? The sort time clearly depends on how many numbers are being sorted.
2. What is the initial order? An examination of Program 7 reveals that it takes less time to sort $n$ numbers that are already in nondecreasing order than when they are not.
3. What computer is the program going to be run on? The time is less on a fast computer and more on a slow one.
4. What programming language and compiler will be used? These influence the quality of the computer code generated for the algorithm.

To resolve the first two questions, we ask for the worst-case or average time as a function of instance size. The worst-case time for any instance size $n$ is defined as

$$t_W(n) = \max\{t(I) \mid I \text{ is an instance of size } n\}$$

Here $t(I)$ denotes the time required for instance $I$. The average time is defined as

$$t_A(n) = \frac{1}{N}\sum_I t(I)$$

where the sum is taken over all instances of size $n$ and $N$ is the number of such instances. In the sorting problem, we can restrict ourselves to the $n!$ different permutations of any $n$ distinct numbers. So $N = n!$.

## A. Analysis

We can avoid answering the last two questions by requiring a rough count of the number of steps executed in the worst or average case rather than an exact time. When this is done, a paper-and-pencil analysis of the algorithm is performed. This is called *performance analysis*.

Let us carry out a performance analysis on our two sorting algorithms. Assume that we wish to determine the worst-case step count for each. Before we can start, we must decide the parameters with respect to which we shall perform the analysis. In our case, we shall obtain times as a function of the number $n$ of numbers to be sorted. First consider insertion sort. Let $t(n)$ denote the worst-case step count of Program 6. If $n \le 1$, then only one step is executed (verify that $n \le 1$). When $n > 1$, the recursive call to $sort(n - 1)$ requires $t(n - 1)$ steps in the worst case and the remaining steps count for some linear function of $n$ step executions in the worst case. The worst case is seen to arise when $x[n]$ is to be inserted into position 1. As a result of this analysis, we obtain the following recurrence for insertion sort,

$$t(n) = \begin{cases} a, & n \le 1 \\ t(n-1) + bn + c, & n > 1 \end{cases}$$

where $a$, $b$, and $c$ are constants. This recurrence can be solved by standard methods for the solution of recurrences.

For merge sort, we see from Program 10 that when $n \le 1$ only a constant number of steps are executed. When $n > 1$, two calls to *MergeSort* and one to *merge* are made. While we have not said much about how the division into $A$ and $B$ is to be performed, this can be done in a constant amount of time. The recurrence for *MergeSort* is now seen to be

$$t(n) = \begin{cases} a, & n \le 1 \\ t(\lfloor n/2 \rfloor) + t(n - \lfloor n/2 \rfloor) & \\ \quad + m(n) + b, & n > 1 \end{cases}$$

where $a$ and $b$ are constants and $m(n)$ denotes the worst-case number of steps needed to merge $n$ numbers. Solving this recurrence is complicated by the presence of the floor function. A solution for the case $n$ is a power of 2 is easily obtained using standard methods. In this case, the floor function can be dropped to get the recurrence

$$t(n) = \begin{cases} a, & n \le 1 \\ 2t(n/2) + m(n) + b, & n > 1 \end{cases}$$

The notion of a step is quite imprecise. It denotes any amount of computing that is not a function of the parameters (in our case $n$). Consequently, a good approximate solution to the recurrences is as meaningful as an exact solution. Since approximate solutions are often easier to obtain than exact ones, we develop a notation for approximate solutions.

**Definition [big "oh"].** $f(n) = O(g(n))$ (read as "$f$ of $n$ is big oh of $g$ of $n$") iff there exist positive constants $c$ and $n_0$ such that $f(n) \le cg(n)$ for all $n$, $n \ge n_0$. Intuitively, $O(g(n))$ represents all functions $f(n)$ whose rate of growth is no more than that of $g(n)$.

EXAMPLE. $3n + 2 = O(n)$ as $3n + 2 \le 4n$ for all $n \ge 2$. $3n + 3 = O(n)$ as $3n + 3 \le 4n$ for all $n \ge 3$. $100n + 6 = O(n)$ as $100n + 6 \le 101n$ for $n \ge 10$. $10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \le 11n^2$ for $n \ge 5$. $1000n^2 + 100n - 6 = O(n^2)$ as $1000n^2 + 100n - 6 \le 1001n^2$ for $n \ge 100$. $6*2^n + n^2 = O(2^n)$ as $6*2^n + n^2 \le 7*2^n$ for $n \ge 4$. $3n +$

$3 = O(n^2)$ as $3n + 3 \leq 3n^2$ for $n \geq 2$. $10n^2 + 4n + 2 = O(n^4)$ as $10n^2 + 4n + 2 \leq 10n^4$ for $n \geq 2$. $3n + 2 \neq O(1)$ as $3n + 2$ is not less than or equal to $c$ for any constant $c$ and all $n$, $n \geq n_0$. $10n^2 + 4n + 2 \neq O(n)$.

As illustrated by the previous example, the statement $f(n) = O(g(n))$ states only that $g(n)$ is an upper bound on the value of $f(n)$ for all $n$, $n \geq n_0$. It does not say anything about how good this bound is. Notice that $n = O(n^2)$, $n = O(n^{2.5})$, $n = O(n^3)$, $n = O(2^n)$, and so on. In order for the statement $f(n) = O(g(n))$ to be informative, $g(n)$ should be as small a function of $n$ as one can come up with for which $f(n) = O(g(n))$. So while we will often say $3n + 3 = O(n)$, we will almost never say $3n + 3 = O(n^2)$, even though the latter statement is correct.

From the definition of $O$, it should be clear that $f(n) = O(g(n))$ is not the same as $O(g(n)) = f(n)$. In fact, it is meaningless to say that $O(g(n)) = f(n)$. The use of the symbol $=$ is unfortunate because it commonly denotes the "equals" relation. Some of the confusion that results from the use of this symbol (which is standard terminology) can be avoided by reading the symbol $=$ as "is" and not as "equals."

The recurrence for insertion sort can be solved to obtain

$$t(n) = O(n^2)$$

To solve the recurrence for merge sort we must use the fact that $m(n) = O(n)$. Using this, we obtain

$$t(n) = O(n \log n)$$

It can be shown that the average number of steps executed by insertion sort and merge sort are, respectively, $O(n^2)$ and $O(n \log n)$.

Analyses such as those performed above for the worst-case and average times are called *asymptotic analyses*. $O(n^2)$ and $O(n \log n)$ are, respectively, the worst-case asymptotic time complexities of insertion and merge sort. Both represent the behavior of the algorithms when $n$ is suitably large. From this analysis, we learn that the growth rate of the computing time for merge sort is less than that for insertion sort. So even if insertion sort is faster for small $n$, when $n$ becomes suitably large, merge sort will be faster.

While most asymptotic analysis is carried out using the big "oh" notation, analysts have available to them three other notations. These are defined below.

**Definition [omega, theta, and little "oh"].** $f(n) = \Omega(g(n))$ (read as "$f$ of $n$ is omega of $g$ of $n$") iff there exist positive constants $c$ and $n_0$ such that $f(n) \geq cg(n)$ for all $n$, $n \geq n_0$. $f(n)$ is $\Theta(g(n))$ (read as "$f$ of $n$ is theta of $g$ of $n$") iff there exist positive constants $c_1$, $c_2$, and $n_0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n$, $n \geq n_0$. $f(n) = o(g(n))$ (read as "$f$ of $n$ is little oh of $g$ of $n$") iff $\lim_{n\to\infty} f(n)/g(n) = 1$.

EXAMPLE. $3n + 2 = \Omega(n)$; $3n + 2 = \Theta(n)$; $3n + 2 = o(3n)$; $3n^3 = \Omega(n^2)$; $2n^2 + 4n = \Theta(n^2)$; and $4n^3 + 3n^2 = o(4n^3)$.

The omega notation is used to provide a lower bound, while the theta notation is used when the obtained bound is both a lower and an upper bound. The little "oh" notation is a very precise notation that does not find much use in the asymptotic analysis of algorithms. With these additional notations available, the solution to the recurrences for insertion and merge sort are, respectively, $\Theta(n^2)$ and $\Theta(n \log n)$.

The definitions of $O$, $\Omega$, $\Theta$, and $o$ are easily extended to include functions of more than one variable. For example, $f(n, m) = O(g(n, m))$ iff there exist positive constants $c$, $n_0$, and $m_0$ such that $f(n, m) \leq cg(n, m)$ for all $n \geq n_0$ and all $m \geq m_0$.

As in the case of the big "oh" notation, there are several functions $g(n)$ for which $f(n) = \Omega(g(n))$. $g(n)$ is only a lower bound on $f(n)$. The theta notation is more precise than both the big "oh" and omega notations. The following theorem obtains a very useful result about the order of $f(n)$ when $f(n)$ is a polynomial in $n$.

**Theorem 1.** Let $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_0$, $a_m \neq 0$.

(a) $f(n) = O(n^m)$
(b) $f(n) = \Omega(n^m)$
(c) $f(n) = \Theta(n^m)$
(d) $f(n) = o(a_m n^m)$

Asymptotic analysis can also be used for space complexity. While asymptotic analysis does not tell us how many seconds an algorithm will run for or how many words of memory it will require, it does characterize the growth rate of the complexity. If an $\Theta(n^2)$ procedure takes 2 sec when $n = 10$, then we expect it to take ~8 sec when $n = 20$ (i.e., each doubling of $n$ will increase the time by a factor of 4).

We have seen that the time complexity of an algorithm is generally some function of the instance characteristics. As noted above, this function is very useful in determining how the time requirements vary as the instance characteristics change. The complexity function can also be used to compare two algorithms A and B that perform the same task. Assume that algorithm A has complexity $\Theta(n)$ and algorithm B is of complexity $\Theta(n^2)$. We can assert that algorithm A is faster than algorithm B for "sufficiently large" $n$. To see the validity of this assertion, observe that the actual computing time of A is bounded from above by $c^*n$ for some constant $c$ and for all $n$, $n \geq n_1$, while that of B is bounded from below by $d^*n^2$ for some constant $d$ and all $n$, $n \geq n_2$. Since $cn \leq dn^2$ for $n \geq c/d$, algorithm A is faster than algorithm B whenever $n \geq \max\{n_1, n_2, c/d\}$.

We should always be cautiously aware of the presence of the phrase "sufficiently large" in the assertion of the preceding discussion. When deciding which of the two algorithms to use, we must know whether the $n$ we are dealing with is in fact "sufficiently large." If algorithm A actually runs in $10^6n$ milliseconds while algorithm B runs in $n^2$ milliseconds and if we always have $n \leq 10^6$, then algorithm B is the one to use.

Table I and Fig. 1 indicate how various asymptotic functions grow with $n$. As is evident, the function $2^n$ grows very rapidly with $n$. In fact, if a program needs $2^n$ steps for execution, then when $n = 40$ the number of steps needed is $\sim 1.1 \times 10^{12}$. On a computer performing 1 billion steps per second, this would require $\sim 18.3$ min (Table II). If $n = 50$, the same program would run for $\sim 13$ days on this computer. When $n = 60$, $\sim 310.56$ years will be required to execute the program and when $n = 100$, $\sim 4 \times 10^{13}$ years will be needed. So we may conclude that the utility of programs with exponential complexity is limited to small $n$ (typically $n \leq 40$).

Programs that have a complexity that is a polynomial of high degree are also of limited utility. For example, if a program needs $n^{10}$ steps, then using our 1 billion steps per second computer (Table II) we will need 10 sec when $n = 10$; 3171 years when $n = 100$; and $3.17 * 10^{13}$ years when $n = 1000$. If the program's complexity were $n^3$ steps instead, we would need 1 sec when $n = 1000$; 110.67 min when $n = 10{,}000$; and 11.57 days when $n = 100{,}000$.

One should note that currently only the fastest computers can execute $\sim 1$ billion instructions per second. From a practical standpoint, it is evident that for reasonably large $n$ (say $n > 100$) only programs of small complexity (such as $n$, $n \log n$, $n^2$, $n^3$, etc.) are feasible. Furthermore, this would be the case even if one could build a computer capable of executing $10^{12}$ instructions per second. In this case, the computing times of Table II would decrease by a factor of 1000. Now, when $n = 100$, it would take 3.17 years to execute $n^{10}$ instructions, and $4 \times 10^{10}$ years to execute $2^n$ instructions.

TABLE I. Values of Selected Asymptotic Functions

| $\log n$ | $n$ | $n \log n$ | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 2 |
| 1 | 2 | 2 | 4 | 8 | 4 |
| 2 | 4 | 8 | 16 | 64 | 16 |
| 3 | 8 | 24 | 64 | 512 | 256 |
| 4 | 16 | 64 | 256 | 4,096 | 65,536 |
| 5 | 32 | 160 | 1,024 | 32,768 | 4,294,967,296 |

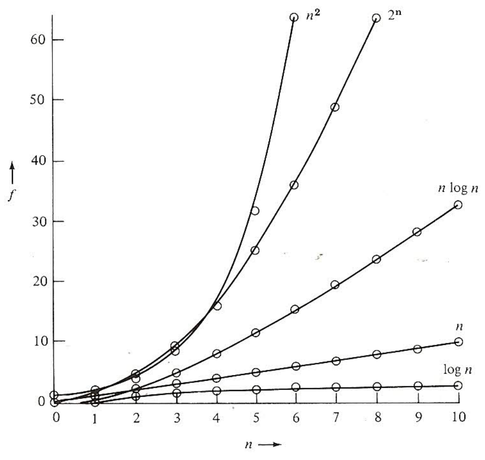

FIG. 1. Plot of selected asymptotic functions.

## B. Measurement

In a performance measurement, actual times are obtained. To do this, we must refine our algorithms into computer programs written in a specific programming language and compile these on a specific computer using a specific compiler. When this is done, the two programs can be given worst-case data (if worst-case times are desired) or average-case data (if average times are desired) and the actual time taken to sort measured for different instance sizes. The generation of worst-case and average test data is itself quite a challenge. From the analysis of Program 7, we know that the worst case for insertion sort arises when the number inserted on each iteration of the **for** loop gets into position 1. The initial sequence ($n$, $n - 1$, ..., 2, 1)

TABLE II. Times on a 1 Billion Instruction per Second Computer[a]

| | Time for $f(n)$ instructions on a $10^9$ instruction/sec computer | | | | | | |
|---|---|---|---|---|---|---|---|
| $n$ | $f(n) = n$ | $f(n) = n \log_2 n$ | $f(n) = n^2$ | $f(n) = n^3$ | $f(n) = n^4$ | $f(n) = n^{10}$ | $f(n) = 2^n$ |
| 10 | 0.01 μsec | 0.03 μsec | 0.1 μsec | 1 μsec | 10 μsec | 10 sec | 1 μsec |
| 20 | 0.02 μsec | 0.09 μsec | 0.4 μsec | 8 μsec | 160 μsec | 2.84 hr | 1 msec |
| 30 | 0.03 μsec | 0.15 μsec | 0.9 μsec | 27 μsec | 810 μsec | 6.83 day | 1 sec |
| 40 | 0.04 μsec | 0.21 μsec | 1.6 μsec | 64 μsec | 2.56 msec | 121.36 day | 18.3 min |
| 50 | 0.05 μsec | 0.28 μsec | 2.5 μsec | 125 μsec | 6.25 msec | 3.1 yr | 13 day |
| 100 | 0.10 μsec | 0.66 μsec | 10 μsec | 1 msec | 100 msec | 3171 yr | $4 \times 10^3$ yr |
| 1,000 | 1.00 μsec | 9.96 μsec | 1 msec | 1 sec | 16.67 min | $3.17 \times 10^3$ yr | $32 \times 10^{283}$ yr |
| 10,000 | 10.00 μsec | 130.3 μsec | 100 msec | 16.67 min | 115.7 day | $3.17 \times 10^{23}$ yr | — |
| 100,000 | 100.00 μsec | 1.66 msec | 10 sec | 11.57 day | 3171 yr | $3.17 \times 10^{33}$ yr | — |
| 1,000,000 | 1.00 msec | 19.92 msec | 16.67 min | 31.71 yr | $3.17 \times 10^7$ yr | $3.17 \times 10^{43}$ yr | — |

[a] 1 μsec = $10^{-6}$ sec; 1 msec = $10^{-3}$ sec.

causes this to happen. This is the worst-case data for Program 7. How about average-case data? This is somewhat harder to arrive at. For the case of merge sort, even the worst-case data are difficult to devise. When it becomes difficult to generate the worst-case or average data, one resorts to simulations.

Suppose we wish to measure the average performance of our two sort algorithms using the programming language Pascal and the TURBO® Pascal (TURBO is a trademark of Borland International) compiler on an IBM-PC. We must first design the experiment. This design process involves determining the different values of $n$ for which the times are to be measured. In addition, we must generate representative data for each $n$. Since there are $n!$ different permutations of $n$ distinct numbers, it is impractical to determine the average run time for any $n$ (other than small $n$'s, say $n < 9$) by measuring the time for all $n!$ permutations and then computing the average. Hence, we must use a reasonable number of permutations and average over these. The measured average sort times obtained from such an experiment are shown in Table III.

As predicted by our earlier analysis, merge sort is faster than insertion sort. In fact, on the average, merge sort will sort 1000 numbers in less time than insertion sort will take for 300! Once we have these measured times, we can fit a curve (a quadratic in the case of insertion sort and an $n \log n$ in the case of merge sort) through them and then use the equation of the curve to predict the average times for values of $n$ for which the times have not been measured. The quadratic growth rate of the insertion sort time and the $n \log n$ growth rate of the merge sort times can be seen clearly by plotting these times as in Fig. 2.

By performing additional experiments, we can determine the effects of the compiler and computer used on the relative performance of the two sort algorithms. We shall provide some comparative times using the VAX 11/780 as the second computer. This popular computer is considerably faster than the IBM-PC and costs ~100 times as much. Our first experiment obtains the average run time of Program 8 (the

TABLE III. Average Times for Merge and Insertion Sort[a]

| $n$ | Merge | Insert |
|---|---|---|
| 0 | 0.027 | 0.032 |
| 10 | 1.524 | 0.775 |
| 20 | 3.700 | 2.253 |
| 30 | 5.587 | 4.430 |
| 40 | 7.800 | 7.275 |
| 50 | 9.892 | 10.892 |
| 60 | 11.947 | 15.013 |
| 70 | 15.893 | 20.000 |
| 80 | 18.217 | 25.450 |
| 90 | 20.417 | 31.767 |
| 100 | 22.950 | 38.325 |
| 200 | 48.475 | 148.300 |
| 300 | 81.600 | 319.657 |
| 400 | 109.829 | 567.629 |
| 500 | 138.033 | 874.600 |
| 600 | 171.167 | — |
| 700 | 199.240 | — |
| 800 | 230.480 | — |
| 900 | 260.100 | — |
| 1000 | 289.450 | — |

[a] Times are in hundredths of a second.

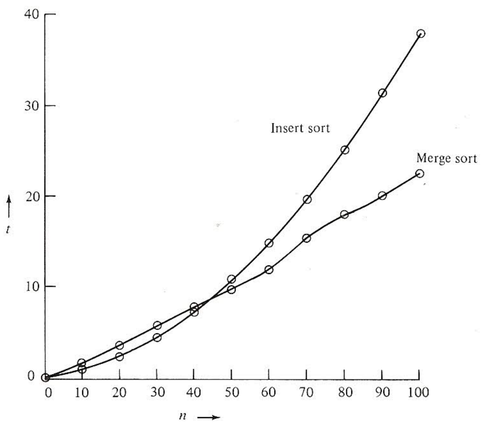

FIG. 2. Plot of times of Table III.

Pascal program for insertion sort). The times for the VAX 11/780 were obtained using the combined translator and interpretive executer, pix. These are shown in Table IV. As can be seen, the IBM-PC outperformed the VAX even though the VAX is many times faster. This is because of the interpreter pix. This comparison is perhaps unfair in that in one case a compiler was used and in the other an interpreter. However, the experiment does point out the potentially devastating effects of using a compiler that generates poor code or of using an interpreter.

In our second experiment, we used the Pascal compiler, pc, on the VAX. This time our insertion sort program ran faster on the VAX. However, as expected, when $n$ becomes suitably large, insertion sort on the VAX is slower than merge sort on an IBM-PC. Sample times are give in Table V. This experiment points out the importance of designing good algorithms. No increase in computer speed can make up for a poor algorithm. An asymptotically faster algorithm will outperform a slower one (when the problem size is suitably large) no matter how fast a computer the slower algorithm is run on and no matter how slow a computer the faster algorithm is run on.

TABLE IV. Average Times for Insertion Sort[a]

| $n$ | IBM-PC turbo | VAX pix |
|---|---|---|
| 50 | 10.9 | 22.1 |
| 100 | 38.3 | 90.47 |
| 200 | 148.3 | 353.9 |
| 300 | 319.7 | 805.6 |
| 400 | 567.6 | 1404.5 |

[a] Times are in hundredths of a second.

TABLE V. Comparison between IBM-PC and VAX[a]

| $n$ | IBM-PC merge sort turbo | VAX insertion sort pc |
|---|---|---|
| 400 | 109.8 | 64.1 |
| 500 | 138.0 | 106.1 |
| 600 | 171.2 | 161.8 |
| 700 | 199.2 | 217.9 |
| 800 | 230.5 | 263.5 |
| 900 | 260.1 | 341.9 |
| 1000 | 289.5 | 418.8 |

[a] Times are in hundredths of a second.

## IV. Lower Bounds

The search for asymptotically fast algorithms is a challenging aspect of algorithm design. Once we have designed an algorithm for a particular problem, we would like to know if this is the asymptotically best algorithm. If not, we would like to know how close we are to the asymptotically best algorithm. To answer these questions, we must determine a function $f(n)$ with the following property:

P1: Let A be any algorithm that solves the given problem. Let its asymptotic complexity be $O(g(n))$. $f(n)$ is such that $g(n) = \Omega(f(n))$.

That is $f(n)$ is a lower bound on the complexity of every algorithm for the given problem. If we develop an algorithm whose complexity is equal to the lower bound for the problem being solved, then the developed algorithm is optimal. The number of input and output data often provides a trivial lower bound on the complexity of many problems. For example, to sort $n$ numbers it is necessary to examine each number at least once. So every sort algorithm must have complexity $\Omega(n)$. This lower bound is not a very good lower bound and can be improved with stronger arguments than the one just used. Some of the methods for obtaining nontrivial lower bounds are the following:

1. Information-theoretic arguments
2. State space arguments
3. Adversary constructions
4. Reducibility constructions

### A. Information-Theoretic Arguments

In an information-theoretic argument one determines the number of different behaviors the algorithm must exhibit in order to work correctly for the given problem. For example, if an algorithm is to sort $n$ numbers, it must be capable of generating $n!$ different permutations of the $n$ input numbers. This is because depending on the particular values of the $n$ numbers to be sorted, any of these $n!$ permutations could represent the right sorted order. The next step is to determine how much time every algorithm that has this many behaviors must spend in the solution of the problem. To determine this quantity, one normally places restrictions on the kinds of computations the algorithm is allowed to perform. For instance, for the sorting problem, we may restrict our attention to algorithms that are permitted to compare the numbers to be sorted but not permitted to perform arithmetic on these numbers. Under these restrictions, it can be shown that $n \log n$ is a lower bound on the average and worst-case complexity of sorting. Since the average and worst-case complexities of merge sort is $\Theta(n \log n)$, we conclude that merge sort is an asymptotically optimal sorting algorithm under both the average and worst-case measures.

Note that it is possible for a problem to have several different algorithms that are asymptotically optimal. Some of these may actually run faster than others. For example, under the above restrictions, there may be two optimal sorting algorithms. Both will have asymptotic complexity $\Theta(n \log n)$. However, one may run in $10n \log n$ time and the other in $20n \log n$ time.

A lower bound $f(n)$ is a tight lower bound for a certain problem if this problem is, in fact, solvable by an algorithm of complexity $O(f(n))$. The lower bound obtained above for the sorting problem is a tight lower bound for algorithms that are restricted to perform only comparisons among the numbers to be sorted.

### B. State Space Arguments

In the case of a state space argument, we define a set of states that any algorithm for a particular problem can be in. For example, suppose we wish to determine the largest of $n$ numbers. Once again, assume we are restricted to algorithms that can perform comparisons among these numbers but cannot perform any arithmetic on them. An algorithm state can be described by a tuple $(i, j)$. An algorithm in this state "knows" that $j$ of the numbers are not candidates for the largest number and that $i = n - j$ of them are. When the algorithm begins, it is in the state $(n, 0)$, and when it terminates, it is in the state $(1, n - 1)$. Let $A$ denote the set of numbers that are candidates for the largest and let $B$ denote the set of numbers that are not. When an algorithm is in state $(i, j)$, there are $i$ numbers in $A$ and $j$ numbers in $B$. The types of comparisons one can perform are $A : A$ ("compare one number in $A$ with another in $A$"), $A : B$, and $B : B$. The possible state changes are as follows:

| | |
|---|---|
| $A : A$ | This results in a transformation from the state $(i, j)$ to the state $(i - 1, j + 1)$. |
| $B : B$ | The state $(i, j)$ is unchanged as a result of this type of comparison. |
| $A : B$ | Depending on the outcome of the comparison, the state either will be unchanged or will become $(i - 1, j + 1)$. |

Having identified the possible state transitions, we must now find the minimum number of transitions needed to go from the initial state to the final state. This is readily seen to be $n - 1$. So every algorithm (that is restricted as above) to find the largest of $n$ numbers must make at least $n - 1$ comparisons.

### C. Adversary and Reducibility Constructions

In an adversary construction, one obtains a problem instance on which the purported algorithm must do at least a certain amount of work if it is to obtain the right answer. This amount of work becomes the lower bound. A reducibility construction is used to show that, employing an algorithm for one problem (A), one can solve another problem (B). If we have a lower bound for problem B, then a lower bound for problem A can be obtained as a result of the above construction.

## V. NP-Hard and NP-Complete Problems

Obtaining good lower bounds on the complexity of a problem is a very difficult task. Such bounds are known for a handful of problems only. It is somewhat easier to relate the complexity of one problem to that of another using the notion of reducibility that we briefly mentioned in the last section. Two very important classes of reducible problems are NP-hard and

NP-complete. Informally, all problems in the class NP-complete have the property that, if one can be solved by an algorithm of polynomial complexity, then all of them can. If an NP-hard problem can be solved by an algorithm of polynomial complexity, then all NP-complete problems can be so solved. The importance of these two classes comes from the following facts:

1. No NP-hard or NP-complete problem is known to be polynomially solvable.
2. The two classes contain more than a thousand problems that have significant application.
3. Algorithms that are not of low-order polynomial complexity are of limited value.
4. It is unlikely that any NP-complete or NP-hard problem is polynomially solvable because of the relationship between these classes and the class of decision problems that can be solved in polynomial nondeterministic time.

We shall elaborate the last item in the following subsections.

### A. Nondeterminism

According to the common notion of an algorithm, the result of every step is uniquely defined. Algorithms with this property are called *deterministic algorithms*. From 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 specific set of possibilities. A computer that executes these operations is allowed to choose any one of these outcomes. This leads to the concept of a *nondeterministic algorithm*. To specify such algorithms we introduce three new functions:

1. **choice**($S$): Arbitrarily choose one of the elements of set S.
2. **failure**: Signals an unsuccessful completion.
3. **success**: Signals a successful completion.

Thus 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.

The computation of a nondeterministic algorithm proceeds in such a way that, whenever there is a set of choices that leads to a successful completion, one such set of choices is made and the algorithm terminates successfully. A nondeterministic algorithm terminates unsuccessfully iff there exists no set of choices leading to a success signal. A computer capable of executing a nondeterministic algorithm in this way is called a nondeterministic computer. (The notion of such a nondeterministic computer is purely theoretical, because no one knows how to build a computer that will execute nondeterministic algorithms in the way just described.)

Consider the problem of searching for an element $x$ in a given set of elements $a[1 \dots n]$, $n \geq 1$. We are required to determine an index $j$ such that $a[j] = x$. If no such $j$ exists (i.e., $x$ is not one of the $a$'s), then $j$ is to be set to 0. A nondeterministic algorithm for this is the following:

```
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 iff there is no $j$ such that $a[j] = x$.

The computing times for **choice**, **success**, and **failure** are taken to be $O(1)$. Thus the above algorithm is of nondeterministic complexity $O(1)$. Note that since $a$ is not ordered, every deterministic search algorithm is of complexity $\Omega(n)$.

Since many choice sequences may lead to a successful termination of a nondeterministic algorithm, the output of such an algorithm working on a given data set may not be uniquely defined. To overcome this difficulty, one normally considers only decision problems, that is, problems with answer 0 or 1 (or true or false). A successful termination yields the output 1, while an unsuccessful termination yields the output 0.

The time required by a nondeterministic algorithm performing on any given input depends on whether there exists a sequence of choices that leads to a successful completion. If such a sequence exists, the time required is the minimum number of steps leading to such a completion. If no choice sequence leads to a successful completion, the algorithm takes $O(1)$ time to make a failure termination.

Nondeterminism appears to be a powerful tool. Program 11 is a nondeterministic algorithm for the sum of subsets problem. In this problem, we are given a multiset, $w(1 \dots n)$ of $n$ natural numbers and another natural number $M$. We are required to determine whether there is a submultiset of these $n$ natural numbers that sums to $M$. The complexity of this nondeterministic algorithm is $O(n)$. The fastest deterministic algo-

PROGRAM 11. Nondeterministic Sum of Subsets Algorithm

```
procedure NonDeterministicSumOfSubsets(W, n, M)
  for i = 1 to n do
    x(i) = choice({0, 1});
  endfor
  if $\sum_{i=1}^{n} w(i)x(i) = M$ then success
  else failure
  endif
end NonDeterministicSumOfSubsets
```

rithm known for this problem has complexity $O(2^{n/2})$.

## B. NP-Hard and NP-Complete Problems

The size of a problem instance is the number of digits needed to represent that instance. An instance of the sum of subsets problem is given by ($w(1)$, $w(2)$, ..., $w(n)$, $M$). If each of these numbers is a positive integer, the instance size is $\lceil \sum_{i=1}^{n} \log_2(w(i) + 1)\rceil + \lceil \log_2(M + 1)\rceil$ if binary digits are used. An algorithm is of polynomial time complexity iff its computing time is $O(p(m))$ for every input of size $m$ and some fixed polynomial $p()$.

Let $P$ be the set of all decision problems that can be solved in deterministic polynomial time. Let $NP$ be the set of decision problems solvable in polynomial time by nondeterministic algorithms. Clearly, $P \subseteq NP$. It is not known whether $P = NP$ or $P \neq NP$. The $P = NP$ problem is important because it is related to the complexity of many interesting problems. There exist many problems that cannot be solved in polynomial time unless $P = NP$. Since, intuitively, one expects that $P \neq NP$, these problems are in "all probability" not solvable in polynomial time. The first problem that was shown to be related to the $P = NP$ problem, in this way, was the problem of determining whether a propositional formula is satisfiable. This problem is referred to as the *satisfiability problem.*

**Theorem 2. Satisfiability is in $P$ iff $P = NP$.** Let A and B be two problems. Problem A is polynomially reducible to problem B (abbreviated A reduces to B, and written as A $\alpha$ B) iff the existence of a deterministic polynomial time algorithm for B implies the existence of a deterministic polynomial time algorithm for A. Thus if A $\alpha$ B and B is polynomially solvable, then so also is A. A problem A is NP-hard iff satisfiability $\alpha$ A. An NP-hard problem A is NP-complete iff A $\in NP$.

Observe that the relation $\alpha$ is transitive (i.e., if A $\alpha$ B and B $\alpha$ C, then A $\alpha$ C). Consequently, if A $\alpha$ B and satisfiability $\alpha$ A then B is NP-hard. So, to show that any problem B is NP-hard, we need merely show that A $\alpha$ B, where A is any known NP-hard problem. Some of the known NP-hard problems are as follows:

NP1: Sum of subsets
*Input:* Multiset $W = \{w_i \mid 1 \le i \le n\}$ of natural numbers and another natural number $M$.
*Output:* "Yes" if there is a submultiset of $W$ that sums to $M$; "No" otherwise.

NP2: 0/1-Knapsack
*Input:* Multisets $P = \{p_i \mid \le i \le n\}$ and $W = \{w_i \mid 1 \le i \le n\}$ of natural numbers and another natural number $M$.
*Output:* $x_i \in \{0, 1\}$ such that $\sum_i p_i x_i$ is maximized and $\sum_i w_i x_i \le M$.

NP3: Traveling salesman
*Input:* A set of $n$ points and distances $d(i, j)$. $d(i, j)$ is the distance between the points $i$ and $j$.
*Output:* A minimum-length tour that goes through each of the $n$ points exactly once and returns to the start of the tour. The length of a tour is the sum of the distances between consecutive points on the tour. For example, the tour $1 \to 3 \to 2 \to 4 \to 1$ has the length $d(1, 3) + d(3, 2) + d(2, 4) + d(4, 1)$.

NP4: Chromatic number
*Input:* An undirected graph $G = (V, E)$.
*Output:* A natural number $k$ such that $k$ is the smallest number of colors needed to color the graph. A coloring of a graph is an assignment of colors to the vertices in such a way that no two vertices that are connected by an edge are assigned the same color.

NP5: Clique
*Input:* An undirected graph $G = (V, E)$ and a natural number $k$.
*Output:* "Yes" if $G$ contains a clique of size $k$ (i.e., a subset $U \subseteq V$ of size $k$ such that every two vertices in $U$ are connected by an edge in $E$) or more. "No" otherwise.

NP6: Independent set
*Input:* An undirected graph $G = (V, E)$ and a natural number $k$.
*Output:* "Yes" if $G$ contains an independent set of size $k$ (i.e., a subset $U \subseteq V$ of size $k$ such that

no two vertices in $U$ are connected by an edge in $E$) or more. "No" otherwise.

NP7: Hamiltonian cycle
*Input:* An undirected graph $G = (V, E)$.
*Output:* "Yes" if $G$ contains a Hamiltonian cycle (i.e., a path that goes through each vertex of $G$ exactly once and then returns to the start vertex of the path). "No" otherwise.

NP8: Bin packing
*Input:* A set of $n$ objects, each of size $s(i)$, $1 \leq i \leq n$ [$s(i)$ is a positive number], and two natural numbers $k$ and $C$.
*Output:* "Yes" if the $n$ objects can be packed into at most $k$ bins of size $C$. "No" otherwise. When packing objects into bins, it is not permissible to split an object over two or more bins.

NP9: Set packing
*Input:* A collection $S$ of finite sets and a natural number $k$.
*Output:* "Yes" if $S$ contains at least $k$ mutually disjoint sets. "No" otherwise.

NP10: Hitting set
*Input:* A collection $S$ of subsets of a finite set $U$ and a natural number $k$.
*Output:* "Yes" if there is a subset $V$ of $U$ such that $V$ has at most $k$ elements and $V$ contains at least one element from each of the subsets in $S$. "No" otherwise.

The importance of showing that a problem A is NP-hard lies in the $P = NP$ problem. Since we do not expect that $P = NP$, we do not expect NP-hard problems to be solvable by algorithms with a worst-case complexity that is polynomial in the size of the problem instance. From Table II, it is apparent that, if a problem cannot be solved in polynomial time (in particular low-order polynomial time), it is intractable, for all practical purposes. If A is NP-complete and if it does turn out that $P = NP$, then A will be polynomially solvable. However, if A is only NP-hard, it is possible for $P$ to equal $NP$ and for A not to be in $P$.

## VI. Coping with Complexity

An optimization problem is a problem in which one wishes to optimize (i.e., maximize or minimize) an optimization function $f(x)$ subject to certain constraints $C(x)$. For example, the NP-hard problem NP2 (0/1-knapsack) is an optimization problem. Here, we wish to optimize (in this case maximize) the function $f(x) = \sum_{i=1}^{n} p_i x_i$ subject to the following constraints:

(a) $x_i \in \{0, 1\}$, $1 \leq i \leq n$
(b) $\sum_{i=1}^{n} w_i x_i \leq M$

A feasible solution is any solution that satisfies the constraints $C(x)$. For the 0/1-knapsack problem, any assignment of values to the $x_i$'s that satisfies constraints (a) and (b) above is a feasible solution. An optimal solution is a feasible solution that results in an optimal (maximum in the case of the 0/1-knapsack problem) value for the optimization function.

There are many interesting and important optimization problems for which the fastest algorithms known are impractical. Many of these problems are, in fact, known to be NP-hard. The following are some of the common strategies adopted when one is unable to develop a practically useful algorithm for a given optimization:

1. Arrive at an algorithm that always finds optimal solutions. The complexity of this algorithm is such that it is computationally feasible for "most" of the instances people want to solve. Such an algorithm is called a *usually good algorithm*. The simplex algorithm for linear programming is a good example of a usually good algorithm. Its worst-case complexity is exponential. However, it can solve most of the instances given it in a "reasonable" amount of time (much less than the worst-case time).
2. Obtain a computationally feasible algorithm that "almost" always finds optimal solutions. At other times, the solution found may have a value very distant from the optimal value. An algorithm with this property is called a *probabilistically good algorithm*.
3. Obtain a computationally feasible algorithm that obtains "reasonably" good feasible solutions. Algorithms with this property are called *heuristic algorithms*. If the heuristic algorithm is guaranteed to find feasible solutions that have value within a prespecified amount of the optimal value, the algorithm is called an *approximation algorithm*.

In the remainder of this section, we elaborate on approximation algorithms and other heuristics.

### A. Approximation Algorithms

When evaluating an approximation algorithm, one considers two measures: algorithm complexity and the quality of the answer (i.e., how close it is to being optimal). As in the case of complexity, the second measure may refer to the worst case or the average case.

There are several categories of approximation algorithms. Let $A$ be an algorithm that generates a feasible solution to every instance $I$ of a problem $P$. Let $F^*(I)$ be the value of an optimal solution, and let $F'(I)$ be the value of the solution generated by $A$.

**Definition.** $A$ is a $k$-absolute approximation algorithm for $P$ iff $|F^*(I) - F'(I)| \leq k$ for all instances $I$. $k$ is a constant. $A$ is an $f(n)$-approximate algorithm for $P$ iff $|F^*(I) - F'(I)|/F^*(I) \leq f(n)$ for all $I$. $n$ is the size of $I$ and we assume that $|F^*(I)| > 0$. An $f(n)$-approximate algorithm with $f(n) \leq \varepsilon$ for all $n$ and some constant $\varepsilon$ is an $\varepsilon$-approximate algorithm.

**Definition.** Let $A(\varepsilon)$ be a family of algorithms that obtain a feasible solution for every instance $I$ of $P$. Let $n$ be the size of $I$. $A(\varepsilon)$ is an approximation scheme for $P$ iff for every $\varepsilon > 0$ and every instance $I$, $|F^*(I) - F'(I)|/F^*(I) \leq \varepsilon$. An approximation scheme whose time complexity is polynomial in $n$ is a polynomial time approximation scheme. A fully polynomial time approximation scheme is an approximation scheme whose time complexity is polynomial in $n$ and $1/\varepsilon$.

For most NP-hard problems, the problem of finding $k$-absolute approximations is also NP-hard. As an example, consider problem NP2 (0/1-knapsack). From any instance $(p_i, w_i, 1 \leq i \leq n, M)$, we can construct, in linear time, the instance $((k + 1)p_i, w_i, 1 \leq i \leq n, M)$. This new instance has the same feasible solutions as the old. However, the values of the feasible solutions to the new instance are multiples of $k + 1$. Consequently, every $k$-absolute approximate solution to the new instance is an optimal solution for both the new and the old instance. Hence, $k$-absolute approximate solutions to the 0/1-knapsack problem cannot be found any faster than optimal solutions.

For several NP-hard problems, the $\varepsilon$-approximation problem is also known to be NP-hard. For others fast $\varepsilon$-approximation algorithms are known. As an example, we consider the optimization version of the bin packing problem (NP8). This differs from NP8 in that the number of bins, $k$, is not part of the input. Instead, we are to find a packing of the $n$ objects into bins of size $C$ using the fewest number of bins. Some fast heuristics that are also $\varepsilon$-approximate algorithms are the following:

*First Fit* (*FF*). Objects are considered for packing in the order 1, 2, ..., $n$. We assume a large number of bins arranged left to right. Object $i$ is packed into the leftmost bin into which it fits.

*Best Fit* (*BF*). Let $cAvail[j]$ denote the capacity available in bin $j$. Initially, this is $C$ for all bins. Object $i$ is packed into the bin with the least $cAvail$ that is at least $s(i)$.

*First Fit Decreasing* (*FFD*). This is the same as FF except that the objects are first reordered so that $s(i) \geq s(i + 1)$, $1 \leq i < n$.

*Best Fit Decreasing* (*BFD*). This is the same as BF except that the objects are reordered as for FFD.

It should be possible to show that none of these methods guarantees optimal packings. All four are intuitively appealing and can be expected to perform well in practice. Let $I$ be any instance of the bin packing problem. Let $b(I)$ be the number of bins used by an optimal packing. It can be shown that the number of bins used by FF and BF never exceeds $(17/10)b(I) + 2$, while that used by FFD and BFD does not exceed $(11/9)b(I) + 4$.

**Example.** Four objects with $s(1:4) = (3, 5, 2, 4)$ are to be packed in bins of size 7. When FF is used, object 1 goes into bin 1 and object 2 into bin 2. Object 3 fits into the first bin and is placed there. Object 4 does not fit into either of the two bins used so far and a new bin is used. The solution produced utilizes 3 bins and has objects 1 and 3 in bin 1, object 2 in bin 2, and object 4 in bin 3.

When BF is used, objects 1 and 2 get into bins 1 and 2, respectively. Object 3 gets into bin 2, since this provides a better fit than bin 1. Object 4 now fits into bin 1. The packing obtained uses only two bins and has objects 1 and 4 in bin 1 and objects 2 and 3 in bin 2.

For FFD and BFD, the objects are packed in the order 2, 4, 1, 3. In both cases a two-bin packing is obtained. Objects 2 and 3 are in bin 1 and objects 1 and 4 in bin 2.

Approximation schemes (in particular fully polynomial time approximation schemes) are also known for several NP-hard problems. We will not provide any examples here.

## B. Other Heuristics

Often, the heuristics one is able to devise for a problem are not guaranteed to produce solutions with value close to optimal. The virtue of these

PROGRAM 12. General Form of an Exchange Heuristic

```
Step 1  Let $i$ be a random feasible solution [i.e.,
        $C(i)$ is satisfied] to the given problem.
Step 2  Perform perturbations (i.e., exchanges) on $i$
        until it is not possible to improve $i$ by such
        a perturbation.
Step 3  Output $i$.
```

heuristics lies in their capacity to produce good solutions most of the time. A general category of heuristics that enjoys this property is the class of exchange heuristics. In an exchange heuristic for an optimization problem, we generally begin with a feasible solution and change parts of it in an attempt to improve its value. This change in the feasible solution is called a perturbation. The initial feasible solution can be obtained using some other heuristic method or may be a randomly generated solution.

Suppose that we wish to minimize the objective function $f(i)$ subject to the constraints $C$. Here, $i$ denotes a feasible solution (i.e., one that satisfies $C$). Classical exchange heuristics follow the steps given in Program 12. This assumes that we start with a random feasible solution. We may, at times, start with a solution constructed by some other heuristic. The quality of the solution obtained using Program 12 can be improved by running this program several times. Each time, a different starting solution is used. The best of the solutions produced by the program is used as the final solution.

## A Monte Carlo Improvement Method

In practice, the quality of the solution produced by an exchange heuristic is enhanced if the heuristic occasionally accepts exchanges that produce a feasible solution with increased $f(\ )$. (Recall that $f$ is the function we wish to minimize.) This is justified on the grounds that a bad exchange now may lead to a better solution later.

In order to implement this strategy of occasionally accepting bad exchanges, we need a probability function prob($i$, $j$) that provides the probability with which an exchange that transforms solution $i$ into the inferior solution $j$ is to be accepted. Once we have this probability function, the Monte Carlo improvement method results in exchange heuristics taking the form given in Program 13. This form was proposed by N. Metropolis in 1953. The variables *counter* and $n$ are used to stop the procedure. If $n$ successive attempts to perform an exchange on $i$ are rejected, then an optimum with respect to the exchange heuristic is assumed to have been reached and the algorithm terminates.

Several modifications of the basic Metropolis scheme have been proposed. One of these is to use a sequence of different probability functions. The first in this sequence is used initially, then we move to the next function, and so on. The transition from one function to the next can be made whenever sufficient computer time has been spent at one function or when a sufficient number of perturbations have failed to improve the current solution. When the sequence of probability functions used is of the form $e^{[f(i)-f(j)]/Y_q}$ for $q = 1, 2, 3, \ldots$, the method is called simulated annealing. The $Y_q$'s are called temperatures. The number of temperatures to use and their values must be determined experimentally. Generally, one starts at a sufficiently high temperature so that almost all perturbations (good and bad) are accepted. This corresponds to the physical process of melting a metal. The temperature is then gradually reduced. At each temperature, enough time is spent to bring the solution to a stable state. The process terminates when the temperature has become small enough that the solution has "frozen" (i.e., random perturbations are no longer accepted).

Several modifications of the simulated annealing scheme described above have been proposed. Much experimentation with simulated annealing has shown that this method can be used to obtain reasonably good solutions if one is willing to invest a large amount of computer time. However, clever heuristics designed for the specific problem to be solved often obtain

PROGRAM 13. Metropolis Monte Carlo Method

```
Step 1  Let $i$ be a random feasible solution to the
        given problem. Set counter = 0.
Step 2  Let $j$ be a feasible solution that is obtained
        from $i$ as a result of a random perturbation.
Step 3  If $f(j) < f(i)$, then [$i = j$, update best
        solution found so far in case $i$ is best,
        counter = 0, go to Step 2].
Step 4  [$f(j) \geq f(i)$] If counter = $n$ then output
        best solution found and stop.
        Otherwise, $r$ = random number in the
        range (0, 1);
        If $r <$ prob($i$, $j$) then [$i = j$, counter = 0]
        else [counter = counter + 1].
        go to Step 2.
```

better solutions using much less computer time. The most significant advantage of simulated annealing appears to be its generality. It can be used on a very wide variety of combinatorial optimization problems and, given enough computer time, one can expect solutions comparable to those produced by heuristics tailored to the specific problem.

## VII. Summary

In order to solve difficult problems in a reasonable amount of time, it is necessary to use a good algorithm, a good compiler, and a fast computer. A typical user, generally, does not have much choice regarding the last two of these. The choice is limited to the compilers and computers the user has access to. However, one has considerable flexibility in the design of the algorithm. Several techniques are available for designing good algorithms and determining how good these are. For the latter, one can carry out an asymptotic analysis. One can also obtain actual run times on the target computer.

When one is unable to obtain a low-order polynomial time algorithm for a given problem, one can attempt to show that the problem is NP-hard or is related to some other problem that is known to be computationally difficult. Regardless of whether one succeeds in this endeavor, it is necessary to develop a practical algorithm to solve the problem. One of the suggested strategies for coping with complexity can be adopted.

## Bibliography

Aho, A., Hopcroft, J., and Ullman, J. (1974). "The Design and Analysis of Computer Algorithms." Addison-Wesley, Reading, Massachusetts.

Garey, M., and Johnson, D. (1979). "Computers and Intractability." Freeman, San Francisco, California.

Horowitz, E., and Sahni, S. (1978). "Fundamentals of Computer Algorithms." Computer Science Press, Rockville, Maryland.

Kirkpatrick, S., Gelatt, C., and Vecchi, M. (1983). Science **220**, 671–680.

Knuth, D. (1972). *Commun. ACM* **15** (7), 671–677.

Nahar, S., Sahni, S., and Shragowitz, E. (1985). *ACM/IEEE Des. Autom. Conf., 1985*, pp. 748–752.

Sahni, S. (1985). "Concepts in Discrete Mathematics." 2nd ed. Camelot Publ. Co., Fridley, Minnesota.

Sahni, S. (1985). "Software Development in Pascal." Camelot Publ. Co., Fridley, Minnesota.

Sedgewick, R. (1983). "Algorithms." Addison-Wesley, Reading, Massachusetts.

Syslo, M., Deo, N., and Kowalik, J. (1983). "Discrete Optimization Algorithms." Prentice-Hall, Englewood Cliffs, New Jersey.