*Acta Cryst.* (1977). **33**, 5–12

# Algorithms and Computational Complexity

BY ALFRED V. AHO

*Bell Laboratories, Murray Hill, New Jersey* 07974, *USA*

(*Received* 29 *March* 1976; *accepted* 23 *July* 1976)

Computer programming was and, in many cases, still is an art rather than a science. Programs are often written without the benefit of any design theory, analysis techniques, or awareness of what others in the field have done. Recently, however, a systematic body of knowledge concerning the design, analysis, and implementation of computer algorithms has begun to emerge. This paper highlights some current developments in this field and shows how proper design techniques can lead to order-of-magnitude improvements in program performance.

## Introduction

As computers are used to attack larger and larger problems, it becomes more and more important to understand how the time and space complexity of the algorithm underlying a computer program affects the size of problem that can be solved by that program. The improvement in program performance achieved by replacing an inefficient algorithm by one of smaller time complexity is often much more spectacular than that achievable just by speeding up the hardware or by rewriting the program in another language.

In this article we discuss recent developments in the algorithm design area which should be of general interest to all computer users. We present some general techniques that can be used to design efficient computer algorithms for many problems. We also present some new theoretical results which suggest that certain classes of frequently encountered combinatorial and optimization problems can require substantial amounts of computation time, regardless of what algorithm is used. These latter results are useful in that they can direct attention to alternative approaches whose computational requirements are less severe.

## Computational complexity

Much of the current interest in algorithms is focused on computational complexity. The basic question asked in complexity theory is: 'How much time and space is required to solve a problem of a given size?'

In theory, the size of a problem is the number of bits of input, with all numbers in binary notation. In practice, several other measures are also used. For example, a convenient measure of size in a sorting problem is the number of items to be sorted, in a transform problem the number of points to be transformed, and in a matrix-inversion problem the dimension of the matrix.

Time is measured in terms of the number of computational steps made in processing an input of size $n$. A computational step is defined as one primitive operation that can be executed with a fixed amount of effort on a computer model such as a Turing machine or a random-access machine.* For our purposes here we can take as a single computational step any operation that can be performed in one instruction on a computer, such as any Boolean operation on two bits, or any arithmetic operation on two integers (provided a fixed bound can be imposed in advance on the size of all integers used).

We define the *worst-case time complexity* of an algorithm to be the maximum number of computational steps required to solve any problem of size $n$, expressed as a function of $n$. The *expected time complexity* is the average number of computational steps, taken over all inputs of size $n$. Obviously, an algorithm with the best worst-case behavior need not have the best expected behavior. Unfortunately, the expected time complexity of an algorithm is often much more difficult to determine, primarily because of the difficulty of coping mathematically with the probability distributions of problems that arise in practice.

When we say an algorithm is of $n^2$ time complexity, the actual running time of any program implementing that algorithm will be bounded from above by $cn^2$ for some positive constant $c$ for inputs of size $n$. Although the constant of proportionality determines the precise running time of a program on a given computer, in this paper we shall leave all constant factors undetermined. There are several reasons for doing this.

First, there is so much hardware variation from one machine to another that a precise value for a constant factor would be meaningful only in a rather limited environment. Second, and more important, it is the functional rate of growth of the complexity rather than the constant factor which is of prime importance in comparing two algorithms. For example, if we have two algorithms, one of time complexity $f(n)$, the other of time complexity $g(n)$, and $f(n)$ grows functionally faster than $g(n)$,† then there always exists a threshold

* See Aho, Hopcroft & Ullman (1974) for definitions and references omitted from this paper, and for a general introduction to the subject of computational complexity.

† This means that there is no constant $c$ such that $f(n) \leq cg(n)$ for sufficiently large $n$. For example, $n \log n$ is functionally greater than $n$ but $n + \log n$ is not.

value $n_0$ of $n$ for which the $g(n)$ algorithm will be faster. When the problem size exceeds this threshold value, the $g(n)$ algorithm will outperform the $f(n)$ algorithm.

### The importance of computational complexity

The practical importance of computational complexity is captured by the following somewhat whimsical example. Two card players $A$ and $B$ play a game in which they need to sort a hand of $n$ cards. Player $A$ sorts his hand in $n$ passes, in each pass removing the largest card and placing it on a pile in front of him (presumably face down). In the $i$th pass he scans $n-i+1$ cards, so in sorting his hand he handles a total of $\sum_{i=1}^{n}(n-i+1)=n(n+1)/2$ cards.

Player $B$ sorts his hand with a 'radix' sort. He makes two passes over his hand. In the first pass he examines each card and places it in one of thirteen piles depending on its rank (*i.e.* whether it is an ace, or a king *etc.*). He then stacks the thirteen piles together, placing one pile on top of the next. In the second pass he places each card in one of four piles depending on its suit (*i.e.* whether it is a spade, a heart *etc.*). After stacking these four piles together, his hand is sorted. We see that player $B$ sorts his hand handling $2n$ cards in all.

Suppose $A$ and $B$ are expert players – each can handle one card in one millisecond. When they are playing with a small number of cards, they can sort their hands in a flash with either algorithm. However, suppose player $B$ (slyly) suggests, 'Let's play two more games, the first with a thousand cards, the second with a million.' Table 1 shows why player $A$ should not accept this invitation.

Table 1. *Time to sort a hand*

| | Number of cards | |
|---|---|---|
| | $10^3$ | $10^6$ |
| $A$ | 9 min | 16 years |
| $B$ | 2 s | 34 min |

Player $A$ obviously needs to reduce the amount of time needed to sort a hand of one million cards. He could obtain a faster sorting machine or he could rewrite his current program in another language. But even if he obtained a machine 1000 times faster than his current one or even if the rewritten program produced a 1000-fold speedup, he would still take 6 days to sort a hand of one million cards.

On the other hand, if he merely replaced his quadratic algorithm with player $B$'s linear algorithm, he could sort one million cards in 34 minutes instead of 16 years.* This example illustrates how important it is, with large amounts of data, to use an algorithm whose time complexity grows as slowly as possible.

* If there are only 52 different card values, player $A$ can do even better by using a single pass and 52 piles.

With small amounts of data, however, the constant of proportionality of the time complexity can be more important than the growth rate itself. For example, suppose we have two algorithms, one whose time complexity is $100n$, the other $10n^2$. Then for all values of $n$ less than 10, the quadratic algorithm would outperform the linear. For inputs of size greater than 10 the linear algorithm becomes the method of choice, and for not-too-large values of $n$ the quadratic algorithm becomes infeasible to use, even on the fastest of machines.

### Algorithm design techniques

It is theoretically impossible to give a general method to find the best algorithm for a given problem, but certain systematic approaches to algorithm design yield good results for large classes of problems. We shall mention a few of the more generally applicable algorithm design techniques here.

*Divide-and-conquer*

Most important, perhaps, is the technique known as 'divide-and-conquer'. Its origins go back to antiquity, but even today it can be used to produce unexpectedly efficient algorithms. In the divide-and-conquer approach, we attempt to solve a given problem by partitioning it into a small set of smaller subproblems whose solutions can be combined to yield a solution to the original problem.

If the same technique is applied recursively to each subproblem, we can easily determine the asymptotic time complexity of the entire algorithm. For example, suppose a problem of size $n$ is partitioned into $a$ subproblems each of size $n/b$. Then $t(n)$, the time complexity of the algorithm for a problem of size $n$, can be expressed in terms of the recurrence

$$t(n)=\begin{cases} k & \text{for } n=1 \\ at(n/b)+cn & \text{for } n>1 \end{cases} \tag{1}$$

where $a$, $b$, $c$ and $k$ are positive constants. Equation (1) assumes it takes $cn$ computational steps to combine the $a$ subproblems of size $n/b$ into a solution to the original problem. Except for multiplicative factors, the solution to equation (1) grows as

$$t(n)\approx\begin{cases} n & \text{if } a<b \\ n\log^2 n & \text{if } a=b \\ n^{\log_b a} & \text{if } a>b. \end{cases} \tag{2}$$

To illustrate the divide-and-conquer technique, let us apply it to sorting, a task in which much computer time is spent. Suppose we sort a sequence of $n$ items by splitting the sequence in the middle, recursively sorting each half by the same method, and then merging the sorted halves. The recurrence governing the time complexity of this process is

$$t(n)=\begin{cases} 1 & \text{for } n=1 \\ 2t(n/2)+n & \text{for } n>1 \end{cases} \tag{3}$$

assuming it takes $n$ computational steps to merge two sorted sequences of length $n/2$. For $n$ a power of two, the solution to equation (3) is

$$t(n) = n \log_2 n + n. \tag{4}$$

This 'merge sort' is vastly superior to any $n^2$ algorithm for large sorting problems.

The principle of balance is often useful in the context of divide-and-conquer. Consider the following 'exchange' algorithm to sort a sequence of $n$ numbers. We find the largest number, exchange it with the last number in the sequence, and then recursively sort the first $n-1$ numbers using the same procedure. Assuming it takes $n$ computational steps to find the largest of $n$ numbers, the time complexity of exchange sort grows quadratically, which as we have seen, is a bad growth rate for a sorting algorithm.

The exchange sort partitions a problem of size $n$ into two smaller subproblems, one of size $n-1$, the other of size 1. The merge sort, on the other hand, uses the principle of balance: it partitions a problem of size $n$ into two subproblems each of size $n/2$. For large values of $n$, the performance of the merge sort is superior to that of the exchange sort.

Binary search is another familiar example of balance and divide-and-conquer. Consider the problem of finding a word in a dictionary of length $n$. If the dictionary is unordered, then on the average we would have to scan $n/2$ words to find the given word. If the dictionary is sorted, however, then by using binary search we can always find a given word in $\log_2 n$ time as follows. We open the dictionary in the middle. If the word is there, we halt. Otherwise, we determine whether the given word is before or after the word in the middle of the dictionary, and recursively apply the same process to the appropriate half of the dictionary.

It is rather surprising that unexpected results can be produced by using as simple a method as divide-and-conquer. For example, divide-and-conquer yields an order $n^{2.81}$ algorithm to multiply two $n \times n$ matrices and an order $n$ algorithm to find the median of $n$ numbers.

*Dynamic programming*

A useful generalization of divide-and-conquer for optimization problems is a method called dynamic programming. Here we attempt to find an optimal solution for a problem by finding optimal solutions for a collection of smaller subproblems. In many situations an efficient algorithm can be obtained by systematically constructing a table of solutions to all subproblems, starting with the smallest subproblems.

A simple example should illustrate the method. Suppose we need to multiply together $n$ matrices $\mathsf{M}_1 \times \mathsf{M}_2 \times \ldots \times \mathsf{M}_n$ where matrix $\mathsf{M}_i$ has $r_{i-1}$ rows and $r_i$ columns for $1 \leq i \leq n$.

The order in which the matrices are multiplied together can dramatically affect the total number of scalar operations (the 'cost'). For example, suppose we need to multiply together three matrices $\mathsf{M}_1$, $\mathsf{M}_2$, and $\mathsf{M}_3$ in which $\mathsf{M}_1$ is $100 \times 1$, $\mathsf{M}_2$ is $1 \times 100$ and $\mathsf{M}_3$ is $100 \times 100$. If we evaluate $(\mathsf{M}_1 \times \mathsf{M}_2) \times \mathsf{M}_3$ in the normal fashion, we require $100 \times 1 \times 100 + 100 \times 100 \times 100 = 1{\cdot}01 \times 10^6$ operations. If we evaluate $\mathsf{M}_1 \times (\mathsf{M}_2 \times \mathsf{M}_3)$, we require only $1 \times 100 \times 100 + 100 \times 1 \times 100 = 20000$ operations.

There are

$$C(n+1) = \frac{1}{n+1}\binom{2n}{n} \simeq \frac{4^n}{n^{3/2}}$$

possible orders in which to multiply a sequence of $n+1$ matrices. [$C(n)$ is the number of different ways in which a sequence of $n$ items can be fully parenthesized. These numbers are called the Catalan numbers.] Thus trying all possible orderings to find the one with minimum cost is an exponential process. Dynamic programming, however, provides an $n^3$ method as follows.

Let $m_{ij}$ be the minimum number of operations needed to evaluate $\mathsf{M}_i \times \mathsf{M}_{i+1} \times \ldots \times \mathsf{M}_j$ for $1 \leq i \leq j \leq n$. We have

$$m_{ij} = \begin{cases} 0, & \text{if } i = j \\ \min\limits_{i \leq k < j} (m_{ik} + m_{k+1,\,j} + r_{i-1} r_k r_j), & \text{if } j > i. \end{cases} \tag{5}$$

Equation (5) finds the optimal way to evaluate $\mathsf{M}_i \times \mathsf{M}_{i+1} \times \ldots \times \mathsf{M}_j$ by considering the costs of the $j-i$ possible products $(\mathsf{M}_i \times \mathsf{M}_{i+1} \times \ldots \times \mathsf{M}_k) \times (\mathsf{M}_{k+1} \times \mathsf{M}_{k+2} \times \ldots \times \mathsf{M}_j)$.

The dynamic programming approach evaluates the $m_{ij}$'s in order of increasing values of $j-i$. We first compute $m_{ii}$ for all $i$, then $m_{i,i+1}$, then $m_{i,i+2}$ and so on until we determine $m_{1n}$, the optimal cost for the $n$-fold product. Choosing this order of evaluation makes sure the terms $m_{ik}$ and $m_{k+1,\,j}$ are available when we evaluate $m_{ij}$ using equation (5). The optimal order to evaluate the matrices can be determined by tracing backward from $m_{1n}$ the values of $k$ used in equation (5).

*Data structures*

One vitally important aspect of algorithm design is the manner in which data is represented inside the computer. The structures holding the data should permit efficient access to elements when they are needed. They should also permit easy insertion and deletion of values when needed. Occasionally these two requirements conflict in that a structure that permits fast access may not permit fast modification. For example, it is easy to find an element in a sorted list, but it is not nearly as easy to add a new element. The principle of balance is often useful in the design of data structures.

Regardless of his specialty, every algorithm designer ought to be familiar with fundamental data structures such as arrays, queues, stacks, and lists (both linear and linked). Hashing and binary trees are important techniques for storing and retrieving data. Basic techniques for representing trees and graphs should also be part of the designer's repertoire. (For details

see Knuth, 1968, 1973; Aho, Hopcroft & Ullman, 1974.)

*Sparse techniques*

Space is often more precious in computing than time. Today a computation requiring a billion computer operations, even floating-point multiplications, is possible but one requiring a billion words of memory is not. For this reason, efficient representations for 'sparse' data are a subject of current research. To illustrate one approach towards sparse data let us consider some representations for the undirected graph $G$ in Fig. 1.

One representation for $G$ is the binary adjacency matrix in Table 2 whose $ij$th element is 1 if, and only if, there is an edge from vertex $i$ to vertex $j$. The adjacency matrix permits easy access to and modification of values but it always requires order $n^2$ space for an $n$-vertex graph.

Table 2. *Adjacency matrix*

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 0 | 1 | 1 | 0 | 0 |
| 3 | 1 | 1 | 0 | 0 | 0 | 0 |
| 4 | 1 | 1 | 0 | 0 | 0 | 0 |
| 5 | 1 | 0 | 0 | 0 | 0 | 1 |
| 6 | 1 | 0 | 0 | 0 | 1 | 0 |

For large values of $n$, the adjacency matrix representation becomes very space consuming. If we take advantage of its symmetry, we can reduce the space requirements by one-half. On the other hand, if the graph is sparse (*i.e.* the number of edges is order $n$ rather than $n^2$), then adjacency lists provide a much more economical representation. Here we store in a linked list for each vertex $i$ only those vertices $j$ such that there is an edge between $i$ and $j$ as in Table 3. The adjacency-lists representation requires order $n$, rather than $n^2$, space for a sparse graph.

Table 3. *Adjacency lists*

| Vertex | | Vertex | |
|---|---|---|---|
| 1 | 2, 3, 4, 5, 6 | 4 | 1, 2 |
| 2 | 1, 3, 4 | 5 | 1, 6 |
| 3 | 1, 2 | 6 | 1, 5 |

*Depth first search*

At the heart of several important algorithms dealing with graphs is a simple efficient technique, called depth first search, for systematically visiting the vertices and edges of a graph $G$. A depth first search begins at some vertex $v$. We then select an untraversed edge $(v,w)$ incident upon $v$. If $w$ has not yet been visited, we move to $w$ and recursively continue the search at $w$. After exhausting all edges incident upon $w$, we return to $v$ and recursively search the remaining untraversed edges incident upon $v$. Fig. 2 shows a depth first search of the graph of Fig. 1, beginning at vertex 1.

The depth first search partitions the edges of $G$ into two sets. The edges by which the vertices (the solid edges in Fig. 2) are reached for the first time are called *tree edges* because they form a spanning tree of $G$. The remaining (dashed) edges link descendants to ancestors and are called *back edges*. These tree and back edges have important mathematical properties which are useful in a number of fundamental graph algorithms, particularly those dealing with connectivity. For example, every cycle in the graph contains at least one back edge.

Depth first search and adjacency lists have been used to solve a variety of graph problems efficiently. One notable application is an order $n$ algorithm to determine whether an $n$-node graph is planar (Hopcroft & Tarjan, 1974). For many years the best known algorithm for this problem had been of time complexity $n^3$.

## Unification of techniques

One of the important benefits of algorithm study is the identification of classes of problems that can be solved with essentially the same algorithmic techniques. Consider, for example, the close relationship between polynomial evaluation and the Fast Fourier Transform.

*Polynomial evaluation*

An $n-1$st degree polynomial

$$p(x) = \sum_{i=0}^{n-1} a_i x^i$$

can be evaluated at $n$ points $c_0, c_1, \ldots, c_{n-1}$ with $n^2$ scalar operations using Horner's rule $n$ times.* On the other hand, consider what happens when we apply divide-and-conquer to this problem. For simplicity, take $n$ to be a power of two. Observe that $p(c)$ is $p(x)$ mod $(x-c)$, *i.e.* the remainder of $p(x)$ divided by $x-c$. Thus we can evaluate $p(x)$ at $c_0, c_1, \ldots, c_{n-1}$ by dividing $p(x)$ by $x-c_0, x-c_1, \ldots, x-c_{n-1}$ and determining the remainders.

* Horner's rule evaluates $p(x)$ as

$$\{(\ldots[(a_{n-1}x + a_{n-2})x + a_{n-3}]x + \ldots)x + a_1\}x + a_0.$$

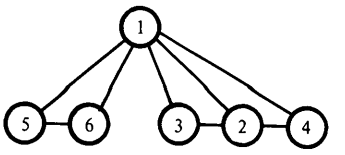


Fig. 1. Undirected graph $G$.

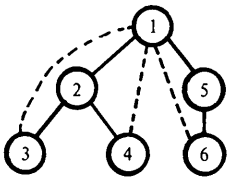


Fig. 2. Depth first search of $G$.

Using a divide-and-conquer approach, we first compute the products

$$p_1(x)=(x-c_0)(x-c_1)\ldots(x-c_{n/2-1}) \quad (6)$$

and

$$p_2(x)=(x-c_{n/2})(x-c_{n/2+1})\ldots(x-c_{n-1}). \quad (7)$$

We now compute $r_1(x)=p(x) \bmod p_1(x)$ and $r_2(x)=p(x) \bmod p_2(x)$. The original problem has now been transformed into two subproblems, each evaluating an $(n-1)/2$ degree polynomial at $n/2$ points: $r_1(x)$ at $c_0, c_1, \ldots, c_{n/2-1}$ and $r_2(x)$ at $c_{n/2}, c_{n/2+1}, \ldots, c_{n-1}$.

When we apply the procedure recursively to the subproblems, the overall time complexity of the process is determined by the recurrence

$$t(n)=\begin{cases} k & \text{for } n=1 \\ 2t(n/2)+d(n) & \text{for } n>1 \end{cases} \quad (8)$$

where $k$ is a constant and $d(n)$ is the time required to divide two $n-1$st degree polynomials. The solution to equation (8) is bounded from above by $d(n)\log n$. Since polynomial division can be done in order $n \log n$ scalar operations, evaluation of an $n-1$st degree polynomial at $n$ points can be performed in order $n \log^2 n$ scalar operations.

This approach to polynomial evaluation can also be used to develop efficient algorithms for a number of other problems such as polynomial interpolation, integer multiplication, and Fourier transform evaluation.

*The Fast Fourier Transform*

The discrete Fourier transform on $n$ points $a_0, a_1, \ldots, a_{n-1}$ is the sequence of values $b_0, b_1, \ldots, b_{n-1}$ where

$$b_j=\sum_{k=0}^{n-1} a_k \omega^{jk}$$

and $\omega$ is a principal $n$th root of unity (*e.g.* $e^{2\pi i/n}$ where $i=\sqrt{-1}$). We see that computing the discrete Fourier transform is equivalent to evaluating the polynomial

$$p(x)=\sum_{i=0}^{n-1} a_i x^i$$

at the roots of unity $\omega^0, \omega^1, \ldots, \omega^{n-1}$.

If we use the polynomial evaluation scheme given above, we immediately have a Fourier transform algorithm that takes $n \log^2 n$ scalar operations. If we notice that $\omega^{n/2}=-1$, however, we can do more. We can rearrange the order in which the output values are to be produced to obtain product polynomials (6) and (7) that have no cross product terms, *i.e.* they are all of the form $x^{2^s}-\omega^{2t}$ for some integers $s$ and $t$.* The advantage of this is that a division by a polynomial of this form can be done in order $n$ time, so in equation (8) $d(n)=cn$ for some constant $c$. The solution to equation (8) then becomes order $n \log_2 n$. These ideas form the basis of the celebrated Fast Fourier Transform (Cooley & Tukey, 1965). Couching these ideas in these terms shows that the same underlying concepts can be applied to a large number of related problems.

## Reducibility

The notion of reducibility carries the concept of algorithm unification further. A problem $P_1$ is said to be *linearly reducible* to a problem $P_2$ if an algorithm for $P_1$ can be constructed out of one for $P_2$ such that the time to perform the reduction and solve $P_1$ is linearly related to the time to solve $P_2$. An analogous definition holds for polynomial time reducibility.

For example, matrix multiplication and matrix inversion are linearly reducible to each other. Similarly, $n \times n$ Boolean matrix multiplication, computing the transitive closure of an $n$-vertex directed graph, and computing the transitive reduction of an $n$-vertex directed graph are all linearly reducible to one another.†

Reducibility is important because it allows results for one problem to be transferred to all problems that are reducible to it. Any improvements to one algorithm would impart improvements to algorithms solving other problems in the same class. Polynomial time reducibility is particularly important both mathematically and computationally. To appreciate the full significance of polynomial time reducibility we need to become acquainted with 'nondeterministic computers' and '*NP*-complete problems'. We use the satisfiability problem from logic to introduce these two concepts.

*The satisfiability problem*

Suppose we are given a Boolean expression $E$ having logical variables, $a, b, c, \ldots$ and we are asked, 'Is there an assignment of the logical variables *true* and *false* to the variables that makes $E$ *true*?' This problem is called the 'satisfiability problem.' One way to solve this problem is to systematically generate all possible assignments of logical values to the variables of $E$ and evaluate $E$ on each.

We can visualize this process as one of generating a solution tree as shown in Fig. 3. From the root of the tree emanate two branches corresponding to the two possible values of variable $a$. Similarly, from each of the two descendants of the root emanate two branches corresponding to the two possible values of variable $b$, and so on. Once we have generated a complete assignment, we evaluate $E$ with that assignment. This

* Choosing $c_j=b_{\mathrm{rev}(j)}$ gives the desired result, where if $[d_{k-1}d_{k-2}\ldots d_0]$ is the binary representation for $j\left(\text{i.e. } j=\sum_{i=0}^{k-1} d_i 2^i\right)$, rev($j$) is the integer whose binary representation is $[d_0 d_1 \ldots d_{k-1}]$.

† $G^*$, the *transitive closure* of directed graph $G$, is a graph with the same vertices as $G$ and with an edge from vertex $i$ to vertex $j$ if, and only if, $G$ has a path from $i$ to $j$. $G^R$, the *transitive reduction* of $G$, is a directed graph with the same vertices as $G$ and with the fewest number of edges of any graph having the same transitive closure as $G$.

evaluation is represented by the path leading up to each leaf.

We designate those leaves at which it is discovered $E$ is *true* 'accepting'. Thus determining whether $E$ is satisfiable is equivalent to determining whether the solution tree has at least one accepting leaf.

One way to determine whether the tree has an accepting leaf is to systematically visit each node of this tree using backtrack programming. Such an approach, however, can be quite time consuming. We can evaluate $E$ on any assignment using a number of computational steps linearly proportional to the length of $E$. Thus the length of the path from the root to a leaf is a linear function of the length of $E$. The number of leaves, however, grows as $2^n$, where $n$ is the number of variables in $E$. This growth rate is so explosive that with $n$ as low as 50, backtrack programming is hopeless, even on a machine that can visit one vertex per nanosecond.

*Nondeterministic computers*

To study problems such as satisfiability, computer scientists employ a fictional machine called a 'nondeterministic' computer. In addition to being able to perform the usual primitive operations of an ordinary (or 'deterministic') computer, a nondeterministic computer has the remarkable capability of being able to split itself, in one step, into several identical copies. Furthermore, each copy has the ability to replicate itself in a similar manner.

We can determine whether there is a solution to the satisfiability problem with a nondeterministic computer as follows. The nondeterministic computer starts off at the root of the solution tree. In one step it replicates itself in two, creating one copy to handle the case $a=true$, the other to handle $a=false$.

In the next step each copy splits in two, one copy handling $b=true$, the other $b=false$. The copies continue operating this way, tracing out the solution tree in parallel, creating new copies to handle each two-way branch. If any copy reaches an accepting leaf, then we know there is a solution to the satisfiability problem.

The time taken by this machine is defined to be the length of a shortest path from the root to an accepting leaf. We see that such a path is of length linear in the size of $E$, so that the nondeterministic computer can determine whether there is a solution to the satisfiability problem in time linear in the size of $E$.

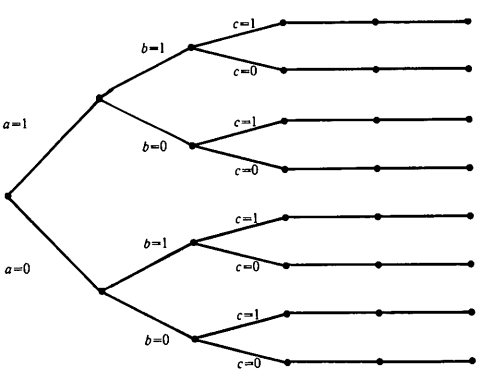


Fig. 3. Solution tree.

We emphasize that a nondeterministic computer is a mathematical abstraction. No real parallel computer model discussed in the literature is capable of coping with the exponential growth needed to implement a nondeterministic computer. The only known way to implement a nondeterministic machine on a real computer is by simulation. This is essentially what backtrack programming does.

*The classes P and NP*

Why study nondeterministic computers? The answer lies in the unity they bring to a large class of important but seemingly disparate problems that arise in many fields of science, engineering and mathematics.

We use $NP$ to denote the class of problems that can be solved on a nondeterministic computer in time polynomial in the size of the problem. (Size here is the number of bits, with numbers in binary notation.) We use $P$ to denote the class of problems that can be solved in polynomial time on a real (deterministic) computer. Obviously $P$ is a subset of $NP$, since a real computer is a special case of a nondeterministic one. No one, however, has yet been able to prove whether $P=NP$ or whether $P$ is a proper subset of $NP$. The answer to this question, as we shall see, would have major implications for the theory of computing.

Certain problems in $NP$ are as hard as any in $P$. We call these problems '$NP$-complete'. Formally, a problem is *$NP$-complete* if it is in $NP$ and every other problem in $NP$ can be polynomially reduced to it.

Cook (1971) was the first to show that $NP$-complete problems exist; in particular, he showed that the satisfiability problem is $NP$-complete. Shortly thereafter, Karp (1972), and later many others, showed that a large number of important and commonly occurring problems are also $NP$-complete. These problems arise in a variety of disciplines: vehicle routing, airplane loading, job-shop scheduling, chemical-compound recognition, optimizing compilers, integer linear programming, and many others.

Here are five well-known examples of $NP$-complete problems. (1) *Traveling salesman.* Given $n$ cities and an integer $k$, is there a circuit that passes through each city once having a total length less than $k$? (2) *Subgraph isomorphism.* Given two graphs $G_1$ and $G_2$, is $G_1$ isomorphic to a subgraph of $G_2$? (3) *Partition.* Given a set of integers, is there a way of partitioning the set into two subsets whose sums are equal? (4) *Graph coloring.* Given a graph $G$ and an integer $k$, can the vertices of $G$ be colored with $k$ colors so that no two adjacent vertices (*i.e.* vertices connected by an edge) have the same color. (5) *Clique.* Given a graph $G$ and an integer $k$, does $G$ contain a $k$-clique (a complete subgraph on $k$ vertices)?

*Implications of NP-completeness*

All *NP*-complete problems are polynomially reducible to one another. Thus, to show a problem is *NP*-complete all we need to do is show that it is in *NP* and that a known *NP*-complete problem is polynomially reducible to it.

The implication of knowing a problem is *NP*-complete is that if a polynomial time-bounded algorithm (on a real computer) existed for that problem, then there would be a polynomial algorithm for every *NP*-complete problem. Although many researchers in a number of different fields have independently studied a variety of *NP*-complete problems, to date no one has found a polynomial algorithm for any *NP*-complete problem. As a consequence it is generally felt that there are no polynomial algorithms for *NP*-complete problems, although no one has yet been able to prove this to be the case. The question of whether there exists a polynomial algorithm for any *NP*-complete problem has become known as the '$P=NP$ problem'* and is considered the major unsolved question in theory of computational complexity.

The practical consequences of discovering a problem to be *NP*-complete have not been fully resolved at present. If indeed $P \neq NP$, no general algorithm can be given which will solve all instances of that problem in less than exponential time. Nevertheless, it may be possible that for the distribution of the instances of the problem encountered in a given application, an algorithm whose expected time complexity is less than exponential can be found. However, the expected time complexity of *NP*-complete problems is an area of active interest that is not yet well understood.

Another possibility is that in practice we may have a restricted form of an *NP*-complete problem. Here we might be able to find an algorithm whose worst-case behavior is less than exponential. Unfortunately, this approach may not always be successful because even rather restricted forms of some *NP*-complete problems are also *NP*-complete. For example, the satisfiability problem is *NP*-complete even if we restrict our attention to Boolean expressions in product-of-sums form with at most three variables per sum [*i.e.* to expressions of the form $(a+b+c)(\bar{a}+c+d)\ldots$. Here, concatenation denotes logical 'and', + logical 'or' and − complementation.] The colorability problem is *NP*-complete even if the graph is planar and has vertices of degree at most 4.

One approach to solving an *NP*-complete problem is to find an algorithm whose exponential growth is confined to some parameter of the problem whose size is small in a given application. For example, generation of an optimal compiler code for expressions with common subexpressions is *NP*-complete even for one-register computers. However, there is an algorithm to generate an optimal code on a one register computer whose time complexity is order $n2^c$ where $n$ is the size of the expressions and $c$ is the number of common subexpressions (Aho, Johnson & Ullman, 1977). In typical expressions the number of common subexpressions is small so this may be a reasonable way to proceed.

Another approach to solving an *NP*-complete problem is to use a heuristic technique that gives a good but not necessarily exact solution. [See, for example, the work of Lin & Kernighan (1973) on the traveling-salesman problem.] For certain classes of problems performance guarantees can be placed on heuristic solutions. For example, consider the problem of 'bin-packing', storing a finite sequence of real numbers between 0 and 1 into the fewest possible number of unit-capacity bins. It has been shown that the heuristic of sorting the sequence into decreasing order and then packing the bins in a first-fit fashion yields a solution that is never worse than 11/9 optimal (Johnson, Demers, Ullman, Garey & Graham, 1974).

Unfortunately, some *NP*-complete problems seem resistant even to heuristic solutions. It has been shown, for example, that if there were a polynomial time-bounded heuristic that always came within a factor of two of determining the chromatic number of a graph,* then $P=NP$ (Garey & Johnson, 1976). Thus it is doubtful that even a good heuristic can be found for graph coloring.

In some cases two seemingly similar problems may have rather dissimilar time complexities. For example, the minimal equivalent of a directed graph $G$ is a smallest subgraph of $G$ that contains the same path information as $G$.† The transitive reduction of $G$ is a smallest graph (not necessarily a subgraph of $G$) that has the same path information as $G$. Finding a minimal equivalent subgraph of $G$ is *NP*-complete (Sahni, 1974). Finding a transitive reduction can be done in less than $n^3$ time. This remark shows how important the precise statement of the problem can be in certain cases.

*Intractable problems*

It is suspected, but not certain, that every *NP*-complete problem requires exponential time in the worst case. There are certain classes of problems for which this is provably the case. Typical of these results is the doubly exponential time complexity of any decision procedure for Presburger arithmetic, a particularly simple system of logic involving only addition and equality. Let $s$ be a first-order predicate calculus statement using the symbols + and =. For example, $s$ might be $\forall x \exists y\ (x+y=y)$. There is a theorem that states that there is a positive constant $c$ and an integer $n_0$ such that for every algorithm to decide whether a first-order statement about Presburger arithmetic is

* If there exists a polynomial time algorithm for one *NP*-complete problem $X$, then every problem in *NP* could be solved in polynomial time by polynomially reducing every problem to $X$. Hence, $P$ would equal *NP*.

* The chromatic number is the least $k$ such that the graph is $k$-colorable.

† If $G'$ is a minimal equivalent subgraph, then there is a path from vertex $i$ to vertex $j$ in $G'$ if, and only if, there is a path from $i$ to $j$ in $G$.

true, and for every $n > n_0$, there is a statement of length $n$ on which the algorithm takes more than $2^{2^{cn}}$ time (Fischer & Rabin, 1974).

This result, and others like it, show that there are theorems whose proofs (in formal sense) are so long that they cannot be found or written down with any reasonable amount of effort. The scope of automatic theorem proving becomes much less ambitious in the light of these results. Exactly which theorems have proofs of reasonable length, however, is still a subject of investigation.

Most discouraging are the results that show that certain problems have no algorithms whatsoever to solve them. Perhaps the most famous of these 'undecidable' problems is the halting problem for Turing machines which, recast in the context of programming, reads as follows. There is no general algorithm which, given any program $P$ and any input $x$, can always determine whether $P$ halts on input $x$. This result becomes even more remarkable when we realize it was proved by Turing (1936) a decade before the first electronic computers were built.

The existence of intractable and unsolvable problems indicates that there are fundamental limits as to which problems can be solved by computers.

## Conclusions

This article has discussed various aspects of algorithms and computational complexity. We have seen that the time complexity of an algorithm determines the ultimate size of problems that can be solved with it. We have discussed some general techniques that can be used to try to design efficient algorithms for a given problem. We have shown that there exist classes of problems for which all attempts at finding general efficient algorithms must fail.

The moral of this paper is, before writing a program to solve a problem, examine the computational complexity of the underlying algorithm. If the program uses an algorithm whose time complexity matches the inherent complexity of the problem, then fruitless searching for nonexistent better algorithms can be avoided. On the other hand, if the program uses an algorithm whose time complexity is functionally greater than is required, then order-of-magnitude improvements in program performance can be achieved by substituting the proper algorithm.

## References


AHO, A. V., HOPCROFT, J. E. & ULLMAN, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison–Wesley.

AHO, A. V., JOHNSON, S. C. & ULLMAN, J. D. (1977). *J. Assoc. Comput. Mach.* **23**, to be published.

COOK, S. A. (1971). *Proc. Third Annual Assoc. for Comput. Mach. Symp. on Theory of Computing*, pp. 151–158.

COOLEY, J. M. & TUKEY, J. W. (1965). *Math. Comput.* **19**, 297–301.

FISCHER, M. J. & RABIN, M. O. (1974). *Complexity of Computation*, edited by R. M. KARP, **7**, 27–42. *SIAM-AMS Proc.*

GAREY, M. R. & JOHNSON, D. S. (1976). *J. Assoc. Comput. Mach.* **23**, 43–49.

HOPCROFT, J. E. & TARJAN, R. E. (1974). *Commun. Assoc. Comput. Mach.* **21**, 549–568.

JOHNSON, D. S., DEMERS, A., ULLMAN, J. D., GAREY, M. R. & GRAHAM, R. L. (1974). *SIAM J. Comput.* **3**, 299–326.

KARP, R. M. (1972). *Complexity of Computer Computations*, edited by R. MILLER & J. THATCHER, pp. 85–104. New York: Plenum Press.

KNUTH, D. E. (1968). *Fundamental Algorithms*. Reading, Mass.: Addison–Wesley.

KNUTH, D. E. (1973). *Sorting and Searching*. Reading, Mass.: Addison–Wesley.

LIN, S. & KERNIGHAN, B. W. (1973). *Oper. Res.* **21**, 498–516.

SAHNI, S. K. (1974). *SIAM J. Comput.* **3**, 262–279.

TURING, A. M. (1936). *Proc. Lond. Math. Soc. Ser.* 2, **42**, 230–265.