Algorithms

An algorithm is a set of rules for getting a specific output from a specific input. Each step must be so precisely defined it can be translated into computer language and executed by machine

by Donald E. Knuth

Ten years ago the word "algorithm" was unknown to most educated people; indeed, it was scarcely necessary. The rapid rise of computer science, which has the study of algorithms as its focal point, has changed all that; the word is now essential. There are several other words that almost, but not quite, capture the concept that is needed: procedure, recipe, process, routine, method, rigmarole. Like these things an algorithm is a set of rules or directions for getting a specific output from a specific input. The distinguishing feature of an algorithm is that all vagueness must be eliminated; the rules must describe operations that are so simple and well defined they can be executed by a machine. Furthermore, an algorithm must always terminate after a finite number of steps.

A program is the statement of an algorithm in some well-defined language. Thus a computer program represents an algorithm, although the algorithm itself is a mental concept that exists independently of any representation. In a similar way the concept of the number 2 exists in our minds without being written down. Anyone who has prepared a computer program will appreciate the fact that an algorithm must be very precisely defined, with an attention to detail that is unusual in comparison with the other things people do.

Programs for numerical problems were written as early as 1800 B.C., when Babylonian mathematicians at the time of Hammurabi gave rules for solving many types of equations. The rules were stated as step-by-step procedures applied systematically to particular numerical examples. The word algorithm itself originated in the Middle East, although at a much later time. It comes from the last name of the Persian scholar Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî, whose textbook on arithmetic (about A.D. 825) had a significant influence for many centuries.

Traditionally algorithms were concerned solely with numerical calculation. Experience with computers has shown, however, that the data manipulated by programs can represent virtually anything. Accordingly the emphasis in computer science has now shifted to the study of various structures by which information can be represented, and to the branching, or decision-making, aspects of algorithms, which allow them to follow one or another sequence of operations depending on the state of affairs at the time. It is precisely these features of algorithms that sometimes make algorithmic models more suitable than traditional mathematical models for the representation and organization of knowledge. Although numerical algorithms certainly have many interesting features, I shall confine the following discussion to non-numerical ones in order to emphasize the fact that algorithms deal primarily with the manipulation of symbols that need not represent numbers.

Searching a Computer's Memory

In order to illustrate how algorithms can fruitfully be studied, I shall consider in some depth a simple problem of retrieving information. The problem is to discover whether or not a certain word, x, appears in a table of words stored in a computer's memory. The word x might be the name of a person, the number of a mechanical part, a word in some foreign language, a chemical compound, a credit-card number or almost anything. The problem is interesting only when the set of all possible x's is too large for the computer to handle all at once; otherwise one could simply set aside one location in the memory for each word.

Suppose n different words have been stored in the computer's memory. The problem is to design an algorithm that will accept as its input the word x and will yield as its output the location j where x appears. Thus the output will be a number between 1 and n, if x is present; on the other hand, if x is not in the memory, the output should be 0, indicating that the search was unsuccessful.

It is, of course, easy to solve this problem. The simplest algorithm is to store the words in locations 1 through n and to look at each word in turn. If x is found in location j, the computer should output j and stop, but if the computer exhausts all n possibilities with no success, it should output 0 and stop. Such a description of the search strategy is probably not precise enough for a computer, however, and so the procedure should be stated more carefully. It might be written as a sequence of steps in the following way:

Algorithm A; sequential search.

A1. [Initialize.] Set $j \leftarrow n$. (The arrow here means that the value of variable j is set equal to n, the number of words in the table to be searched. This is the initial value of j. Subsequent steps of the algorithm will cause j to run through the sequences of values n, $(n - 1)$, $(n - 2)$ and so on until it reaches either 0 or a location containing the input word x.)

A2. [Unsuccessful?] If $j = 0$, output j and terminate the algorithm. (Otherwise go on to step A3.)

A3. [Successful?] If $x = \mathrm{KEY}[j]$, output j and terminate the algorithm. (The term KEY[j] refers to the word stored at location j.)

A4. [Repeat.] Set $j \leftarrow j - 1$ (decrease the value of j by 1) and go back to step A2.

This algorithm can be depicted by a flow chart that may help a person to visualize the steps [*see illustration on page 65*]. One reason it is important to specify the steps carefully is that the algorithm must work in every case. For example, the informal description given first might have suggested an erroneous algorithm that would go directly from step A1 to step A3; such an algorithm would have failed when $n = 0$ (that is, when no words at all were present), since step A1 would set j to 0 and step A3 would refer to the nonexistent KEY[0].

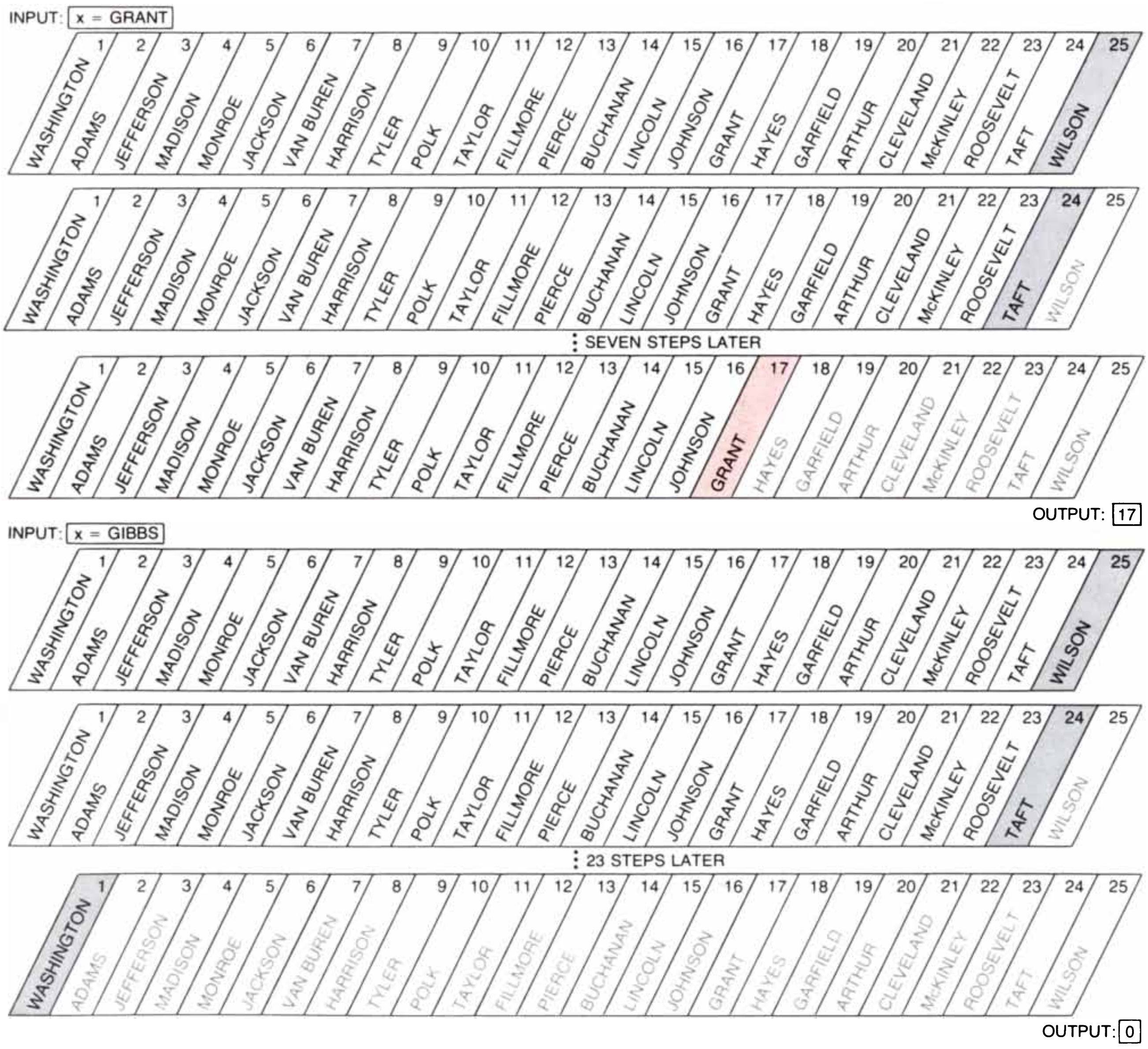
It is interesting to note that Algorithm A can be improved by giving meaning to the notation KEY[0], allowing a word to be stored in "location 0" as well as in locations 1 through n. Then if step A1 sets $\mathrm{KEY}[0] \leftarrow x$ as well as $j \leftarrow n$, step A2 can be eliminated and the search will go about 20 percent faster on many machines. Unfortunately for programmers, the most commonly used computer languages (standard FORTRAN and COBOL) do not allow 0 to be employed as an index for a memory location; thus Algorithm A cannot be so easily improved when it is expressed as a program in those languages.

Algorithm A certainly solves the problem of searching through a table of words, but the solution is not very good unless the number of words to be searched is quite small, say 25 or fewer. If *n* were as large as a million, a simple sequential search would usually be an unbearably slow way to look through the table. We would hardly go to the expense of building such a large table unless we expected to search it frequently, and we would not want to waste any time during the search. Algorithm A is the equivalent of looking for someone's telephone number by going through a telephone directory page by page, column by column, one line at a time. We can do better than that.

The Advantage of Order

It is, in fact, instructive to consider a telephone directory as an example of such a large table of information. If one were asked to find the telephone number of someone who lives at 1642 East 56th Street, there would really be no better way than to do a sequential search equivalent to Algorithm A, since a standard telephone directory is not organized for searches according to address. On the other hand, when one looks up someone's name, it is possible to take advantage of alphabetical order. Alphabetical order is a substantial advantage indeed, since a single glance at almost any point in the directory suffices to eliminate many names from further consideration.

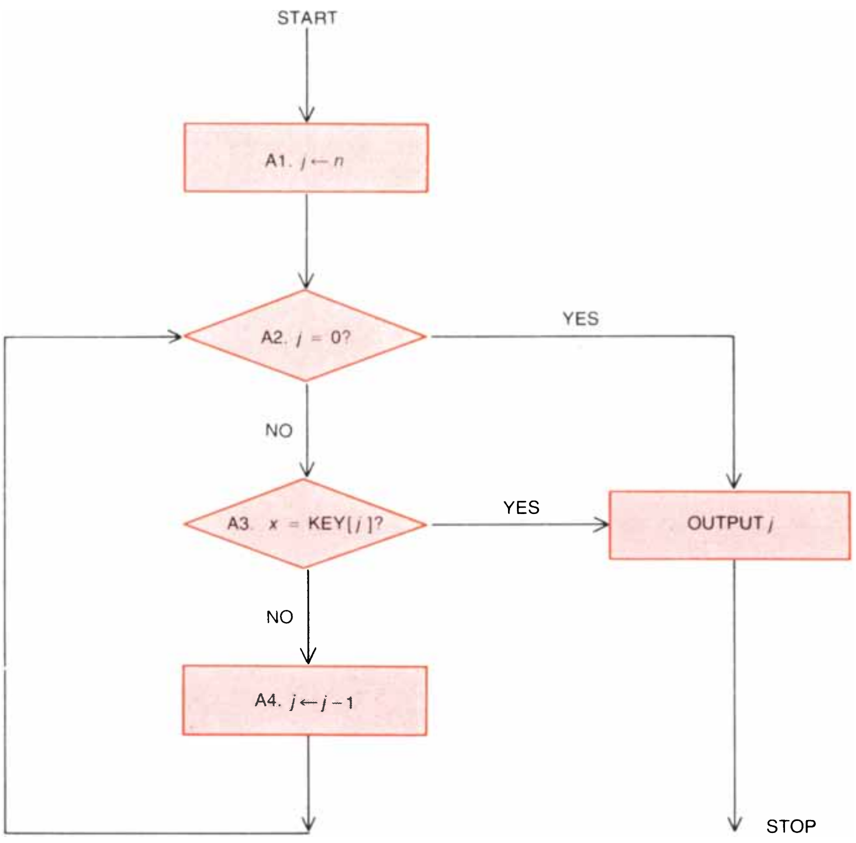
If the words of a table appear consistently in some order, there are several ways to design an efficient search procedure. The simplest procedure starts by looking first at the entry in the middle of



SEQUENTIAL-SEARCH ALGORITHM (Algorithm A in the text of this article) looks for an input word in a table where the entries have not been arranged in any particular order. This table has 25 entries, or keys: KEY[1], KEY[2] and so on up to KEY[25]. Each key is a person's name. Suppose the input word is the name "Grant". Algorithm A searches for "Grant" by comparing it first with KEY[25], which is "Wilson", then with KEY[24], which is "Taft", and so on. Here "Grant" is found to be KEY[17], so that the algorithm outputs "17" (*top*). If input had been "Gibbs", Algorithm A would have compared "Gibbs" with all keys and output would have been 0 (*bottom*).

the table. If the desired word x is numerically or alphabetically less than this middle entry, the entire second half of the table can be eliminated; similarly, if x is greater than the middle entry, one can eliminate the entire first half. Thus a single comparison yields a search problem that is only half as large as the original one. The same technique can now be applied to the remaining half of the table, and so on until the desired word x is either located or proved to be absent. This procedure is commonly known as a binary search.

Although the ideas underlying binary search are simple, some care is necessary in writing the algorithm. First, in a table that has an even number of elements there is no unique "middle" entry. Second, it is not immediately clear when to stop in the case of an unsuccessful search. Teachers of computer science have noticed, in fact, that when students are asked to write a binary-search procedure for the first time, about 80 percent of them get the program wrong, even when they have had more than a year of programming experience! The reader who feels that he understands algorithms fairly well but has never before written a binary-search algorithm might enjoy trying to construct one before reading the following solution.



FLOW CHART FOR ALGORITHM A illustrates the logical path by which the brute-force sequential search looks for an input word x in a table of n keys. The algorithm searches for x by comparing it first with KEY[n], then with KEY[$n-1$], then with KEY[$n-2$] and so on. If x matches some KEY[j], the algorithm outputs j, the location at which x was found. If x is not in the table, the output of the algorithm is 0. Arrow in step A1 ($j \leftarrow n$) means "Set j equal to n" in that step. Step in each box is explained in detail in the fuller form of Algorithm A in third column of text on page 63. On the average Algorithm A must search half of the table to find x. In the worst case, if x is at KEY[1] or if x is not present, Algorithm A must search the entire table.

Algorithm B; binary search. This algorithm employs the same notation as Algorithm A. Moreover, it is assumed that the first word, KEY[1], is less than the second word, KEY[2], which is less than the third word, KEY[3], and so on all the way up to the last word, KEY[n]. This condition can be written $\text{KEY}[1] < \text{KEY}[2] < \text{KEY}[3] < \cdots < \text{KEY}[n]$.

B1. [Initialize.] Set $l \leftarrow 0$, $r \leftarrow n + 1$. (The letter l stands for the left boundary of the search and r stands for the right boundary. More precisely, KEY[j] cannot match the given word x unless the location j is both greater than l and less than r.)

B2. [Find midpoint.] Set $j \leftarrow \lfloor (l + r)/2 \rfloor$. (The brackets $\lfloor\ \rfloor$ mean "Round down to the nearest integer." Thus if $(l + r)$ is even, j is set to $(l + r)/2$; if $(l + r)$ is odd, j is set to $(l + r - 1)/2$.)

B3. [Unsuccessful?] If $j = l$, output 0 and terminate the algorithm. (If j equals l, then r must be equal to $l + 1$, since r is always greater than l; therefore x cannot match any key in the table.)

B4. [Compare.] (At this point $j > l$ and $j < r$.) If $x = \text{KEY}[j]$, output j and terminate the algorithm. If $x < \text{KEY}[j]$, set $r \leftarrow j$ and return to step B2. If $x > \text{KEY}[j]$, set $l \leftarrow j$ and return to step B2.

A play-by-play account of Algorithm B as it searches through a table of 25 names is shown in the illustration on the next page.

It seems clear that binary search (Algorithm B) is much better than sequential search (Algorithm A), but how much better is it? And when is it better? A quantitative analysis will answer these questions.

Quantitative Analysis

First let us analyze the worst cases of algorithms A and B. How long can it possibly take each algorithm to find word x in a table of size n? The answer is easy for Algorithm A. If x equals KEY[1], or if x is not in the table at all, it will take n executions of step A3; that is, the desired word x must be compared with all n entries in the table before the search stops. Furthermore, the algorithm will never execute step A3 more than n times. When sequential search is applied to a table with a million entries, a million comparisons will be made in the worst case.

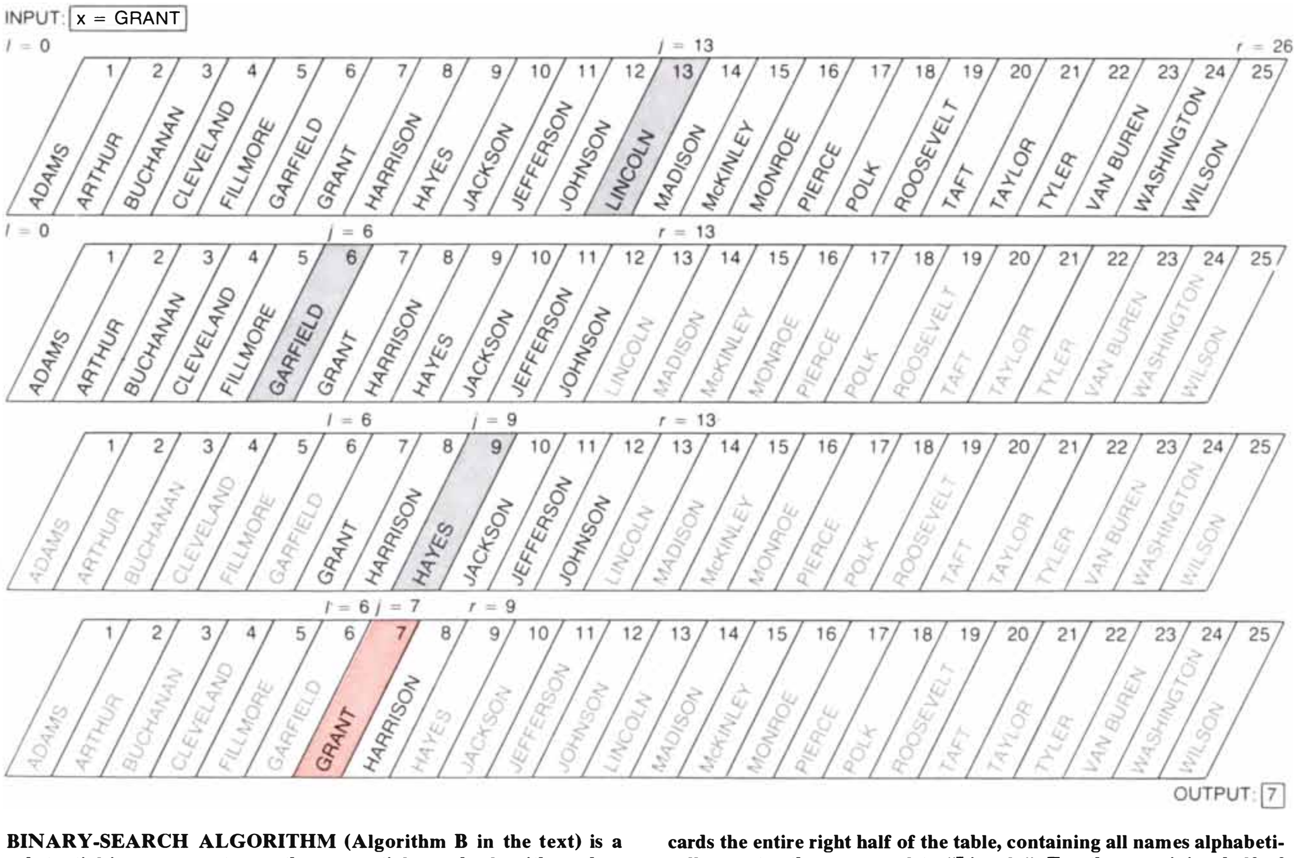
The answer is only slightly more difficult for binary search. Since Algorithm B discards half of the table remaining after each execution of step B4, it first deals with the entire table, then half of the table, then a quarter of the table, then an eighth of the table and so on. The maximum number of executions of step B4 will be k, where k is the smallest integer such that 2^k is greater than n. For example, when binary search is applied to a table with a million (10^6) entries, k will be equal to 20, since 2^{20} is greater than 10^6 but 10^6 is greater than 2^{19}. Thus if a table with 10^6 entries is searched using Algorithm B, at most only 20 of those entries will ever be examined in any particular search.

From the standpoint of worst-case behavior, one can go further and say that Algorithm B is not only a good way to search; it is actually the best possible search algorithm that proceeds solely by comparing x to keys in the table. The reason is that a comparison-based algorithm cannot possibly examine more than $2^k - 1$ different keys during its first k comparisons. No matter what strategy is adopted, the first comparison always selects a particular key of the table and the second comparison will be with at most two other keys (depending on whether x was less than or greater than the first key); the third comparison will be with at most four other keys; the fourth comparison will be with at most eight other keys, and so on. Therefore if a comparison-based search algorithm makes no more than k comparisons, the table can contain no more than $1+2+4+8+\cdots+2^{k-1}$ distinct keys, and this sum equals $2^k - 1$.

The familiar game of Twenty Questions can be analyzed by reasoning in a similar way. In this game one player thinks of a secret object, the name of which he conceals on a folded piece of paper. The other players try to guess what the object is by asking as many as 20 questions that must be answered only by "Yes" or "No." The other players are also initially told whether the secret object is animal, vegetable or mineral, or a combination of those supposedly well-defined attributes. By arguing as I have in the preceding paragraph, one can prove that the other players cannot possibly identify more than 2^{23} different objects correctly, no matter how clever their questions are. There are only 2^3 (or eight) possible subsets of the set of attributes animal, vegetable and mineral, and there are only 2^{20} possible outcomes of the 20 yes-no questions. Thus the total number of objects one can possibly identify is 2^{23}. The argument holds even when each question asked depends on the answers to the preceding questions.

Stating this conclusion another way, if more than 2^{23} different objects must be identified, 20 questions will not always be enough. The search problem is similar but not quite the same, since an algorithm for searching does not simply ask yes-no questions. The questions asked by algorithms of the type we are considering have three possible outcomes, namely $x <$ KEY[j] or $x =$ KEY[j] or $x >$ KEY[j]. When a table contains 2^k or more entries, the above reasoning proves that k comparisons of x with keys in the table will not always be enough. Therefore every algorithm that searches a table of a million words by making comparisons must in some instances examine 20 or more of those words. In short, binary search has the best possible worst case.

The worst-case behavior of an algorithm is not the whole story, since it is overly pessimistic to base decisions entirely on one's knowledge of the worst that can happen. A more meaningful understanding of the relative merits of algorithms A and B can be gained by analyzing their average-case behavior. If each of the n keys in a table is equally likely to be looked up, what is the average number of comparisons that will be needed? For sequential search (Algorithm A) the answer is the simple average $(1+2+3+\cdots+n)/n$, which is equal to $(n+1)/2$. In other words, to find x with Algorithm A one will on the average have to search through about half of the table. To determine the average number of comparisons needed to find x using binary search (Algorithm B), the mathematics is only a little more complicated. In this case the answer is $k - [(2^k - k - 1)/n]$, where k, as before, is the number of comparisons required in the worst case. For large values of n this answer is approximately equal to $k - 1$; therefore the average case of Algorithm B is only about one comparison less than its worst case. By carefully extending the argument made earlier it is possible to show that binary



BINARY-SEARCH ALGORITHM (Algorithm B in the text) is a substantial improvement over the sequential-search algorithm when the table to be searched is large. The entries in the table must first be arranged in order. Here the 25 names are listed in alphabetical order. Again the input word x sought is "Grant". The algorithm compares "Grant" first with the key in the middle location, j, of the table. It calculates the initial value of j by setting the left boundary l of the search at 0 and the right boundary r at $n+1$. In this case r is 26. Then l and r are added together and divided by 2, rounding down to the nearest integer if the answer is not already an integer. The midpoint j of the table is 26/2, or 13, which is the location of "Lincoln" (*top*). Since the name "Grant" is alphabetically less than "Lincoln", the algorithm discards the entire right half of the table, containing all names alphabetically greater than or equal to "Lincoln". For the remaining half of the table the algorithm calculates a new midpoint, first setting r equal to the location j just examined, which is 13 (*second from top*). The new midpoint j is $(0+13)/2$, which must be rounded down to 6, location of "Garfield". "Grant" is alphabetically greater than "Garfield", so that the left quarter of the table is discarded and the left boundary l is set equal to 6 (*second from bottom*). When procedure is repeated once more, "Grant" is found in position 7 (*bottom*). If input word x had been "Gibbs", Algorithm B would have executed one more step, with l still equal to 6 and r set at 7. Midpoint j would have been 6, which is left boundary of search, meaning that "Gibbs" is not in table.

The legend continues...

Introducing the world's most sophisticated Diesel passenger car. The new Mercedes-Benz 300D.

The new, 5-passenger Mercedes-Benz 300D—the state of the Diesel passenger car art.

Here is a most ingenious alternative to the conventional automobile. A truly remarkable new Mercedes-Benz. With a contemporary new look. With ample room for five people, an astonishing 5-cylinder engine and an unusually complete array of luxurious appointments and safety systems. The new 300D. The most sophisticated Diesel passenger car the world has ever seen.

For years, you've heard about exotic and promising alternatives to the conventional automobile engine. To date, only one alternative has kept its promise: the Diesel engine—for over 60 years, the most efficient combustion power plant in use.

Now Mercedes-Benz has synthesized its proven, 5-cylinder Diesel engine with new, technologically advanced body design, suspension, steering and safety systems to produce the most ingenious alternative to the conventional automobile.

A matter of taste

Though only a trim 190.9 inches from bumper to bumper, the new 300D is an honest 5-passenger sedan. The secret of its spaciousness lies in new Mercedes-Benz technology that puts the room in the car *in* the car—without adding bulk or sacrificing safety.

Enter a new 300D and you're surrounded by a complete array of security and convenience features. All are standard equipment. Such things as cruise control, bi-level climate control, electric windows, AM/FM radio, central locking system, 3-speed windshield wipers.

The new 300D is not an exercise in opulence. But it does exhibit meticulous taste. And as your senses will tell you, there's quite a difference between the two ideas.

Sports car handling

The new 300D is one of the most sparkling road cars Mercedes-Benz has ever engineered. Its sophisticated power train, suspension and steering are those of a sports car. And that is why the new 300D handles like one.

The new 300D's unique, 5-cylinder engine is the most powerful, the smoothest Diesel yet engineered into a passenger car. But you pay no penalty for this performance bonus. The EPA estimates that the new 300D should deliver up to 28 mpg on the highway, 23 mpg in town. (Your mileage will depend on how and where you drive and the condition and equipment of your car.)

The state of the art

For over 40 years, Mercedes-Benz has pioneered many of the major advances in Diesel passenger car engineering. The new 300D is the culmination of that experience. It is the state of the Diesel passenger car art.

Test drive the new 300D. Experience the most ingenious alternative to the conventional automobile. The most sophisticated Diesel passenger car in the world.

Mercedes-Benz
Engineered like no other car in the world.

The Small Computer

Twenty-five years ago a computer as powerful as the new Processor Technology Sol-20 priced out at a cool million.

Now for only $995 in kit form or $1495 fully assembled and tested you can have your own small computer with perhaps even more power. It comes in a package about the size of a typewriter. And there's nothing like it on the market today. Not from IBM, Burroughs, DEC, HP or anybody else!

It fills a new role

If you're an engineer, scientist or businessman, the Sol-20 can help you solve many or all of your design problems, help you quantify research, and handle the books too. For not much more than the price of a good calculator, you can have high level computer power.

Use it in the office, lab, plant or home

Sol-20 is a smart terminal for distributed processing. Sol-20 is a stand alone computer for data collection, handling and analysis. Sol-20 is a text editor. In fact, Sol-20 is the key element of a full fledged computer system including hardware, software and peripheral gear. It's a computer system with a keyboard, extra memory, I/O interfaces, factory backup, service notes, users group.

It's a computer you can take home after hours to play or create sophisticated games, do your personal books and taxes, and a whole host of other tasks.

Those of you who are familiar with small computers will recognize what an advance the Sol-20 is.

Sol-20 offers all these features as standard:

8080 microprocessor — 1024 character video display circuitry — control PROM memory — 1024 words of static low-power RAM — 1024 words of preprogrammed PROM — built-in cassette interface capable of controlling two recorders at 1200 bits per second — both parallel and serial standardized interface connectors — a complete power supply including ultra quiet fan — a beautiful case with solid walnut sides — software which includes a preprogrammed PROM personality module and a data cassette with BASIC-5 language plus two sophisticated computer video games — the ability to work with all S-100 bus products.

Full expansion capability

Tailor the Sol-20 system to your applications with our complete line of peripheral products. These include the video monitor, audio cassette and digital tape systems, dual floppy disc system, expansion memories, and interfaces.

Write for our new 22 page catalog. Get all the details.

Processor Technology, Box N, 6200 Hollis St., Emeryville, CA 94608. (415) 652-8080.

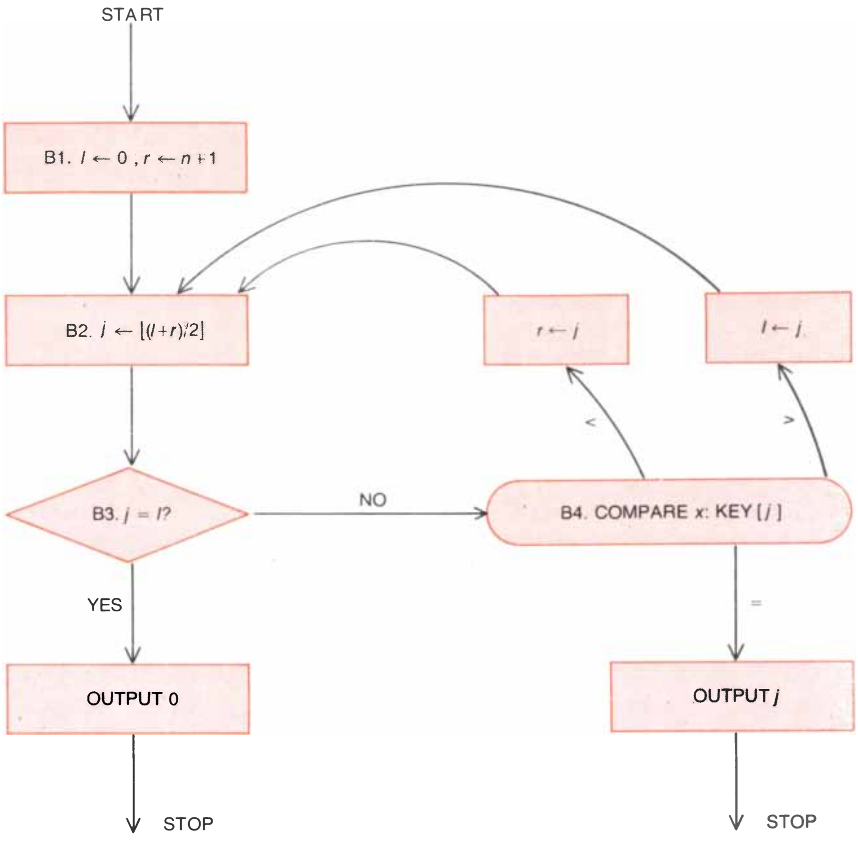
Processor Technology Corporation

search is also the best possible algorithm from the standpoint of the average case: every search algorithm must make at least $k - [(2^k - k - 1)/n]$ comparisons on the average, and many do worse.

Better than the Best

As soon as something has been proved impossible, a lot of people immediately try to do it anyway. This seems to be an inherent component of human behavior. I have just proved that binary search is the best possible way to search a computer's memory, and so naturally I shall now look for a better way.

In the first place, when the number of words in a table is small, Algorithm A actually turns out to be better than Algorithm B. Why does this not contradict the proof that binary search is best? The reason is that in comparing Algorithm A and Algorithm B I have so far been contrasting only the number of comparisons each algorithm makes. Actually Algorithm A requires less bookkeeping activity, so that it takes less time for a machine to execute each comparison. On a typical computer Algorithm A can be made to take about $2n + 6$ units of time, on the average, for a table of size n. Algorithm B, on the other hand, will require an average of about $12 \log_2 n - 11 + 12(k + 1)/n$ units of time, under the same assumptions. Thus unless there are 20 or more keys to be searched, Algorithm A will be better than Algorithm B. These numbers will vary slightly from computer to computer, but they show that the efficiency of an algorithm cannot be determined by counting only the comparisons made.


START
B1. $l \leftarrow 0, r \leftarrow n+1$
B2. $j \leftarrow \lfloor (l+r)/2 \rfloor$
$r \leftarrow j$
$l \leftarrow j$
<
>
B3. $j = l$?
NO
B4. COMPARE x: KEY $[j]$
YES
=
OUTPUT 0
OUTPUT j
STOP
STOP

FLOW CHART OF ALGORITHM B illustrates the rules governing binary search. The algorithm searches for the input word x in a table of n keys that have previously been arranged in order. First x is compared with the middle entry of the table. If x is greater than (>) the middle entry, it is compared with the midpoint of the right half of the table. If x is less than (<) the middle entry, it is compared with the midpoint of the left half of the table. The process continues, with half of the remaining table being discarded each time, until either x is found or the search reveals that x is not in the table. The half brackets (⌊ ⌋) mean "Round down to the nearest integer." Like Algorithm A, Algorithm B is written out in detail in first column of text on page 65.

There is another reason Algorithm B can be beaten. When we look up someone's name in a telephone directory and compare the desired name x with the names on a page, our subsequent action is not influenced solely by whether the comparison shows that the desired name is alphabetically less than or greater than the names on the page; we also observe how much less than or how much greater than, and we turn over a larger chunk of pages if we think we are farther from the goal. The above proof that binary search is best does not apply to algorithms that make use of such things as the degree of difference between x and a particular key. The proof for Twenty Questions can be attacked on similar grounds. For example, the players might notice the length of the secret word as it is being written down, or they might be able to gain information from the length of time the player being questioned hesitates before answering "Yes" or "No."

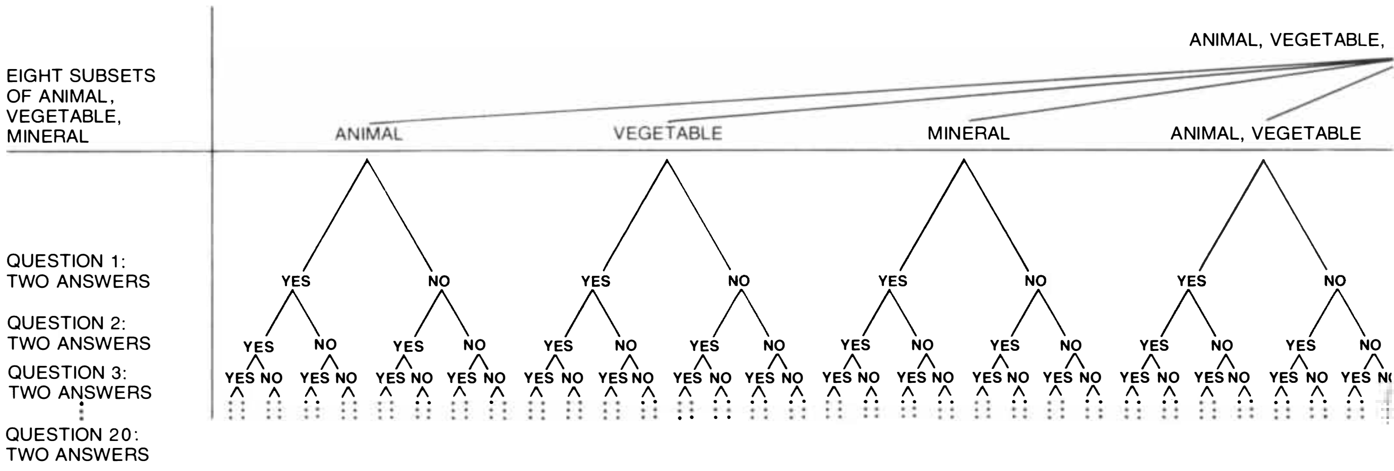
Therefore a human being concerned about efficiency need not begin searching a telephone directory by bisecting it as a computer would; the time-honored method of interpolation with the aid of alphabetical order probably works better in spite of the proof that the binary search is best. In fact, Andrew C. Yao of the Massachusetts Institute of Technology and F. Frances Yao of Brown University have recently shown that the average number of times an interpolation-search algorithm needs to access the table is only $\log_2 \log_2 n$ plus at most a small constant, provided that the table entries are independent and uniformly distributed random numbers. When n is very large, $\log_2 \log_2 n$ is much smaller than $\log_2 n$, so that interpolation search will be significantly faster than binary search. The idea underlying the Yaos' proof is that each iteration of an interpolation search tends to reduce the uncertainty of the position of x from n to the square root of n. Furthermore, they have proved that interpolation search is nearly the best possible, in a very broad sense: any algorithm that searches such a random table by making appropriate comparisons must examine approximately $\log_2 \log_2 n$ entries, on the average.

These results are of great theoretical importance, although computational experience has shown that an interpolation search is usually not an improvement over binary search in practice. The reason is that the data stored in a table are typically not random enough to conform to the assumption of a uniform distribution; in addition n is typically small enough so that the extra calculation per comparison required by each interpolation outweighs the amount of time saved by reducing the number of comparisons. The simplicity of binary search is one of its virtues, and it is important to maintain a proper balance between theory and practice.

Binary Tree Search

The binary search can be improved, however, in another way: by dropping the assumption that every key in the table is equally likely to be sought. When some keys are known to be far more likely candidates than others, an efficient algorithm will examine the more likely ones first.

Before we explore this notion it will be helpful to look first at the binary search in a different way. Consider the 31 words that are used most frequently in the English language (according to Helen Fouché Gaines in her book



THE GAME TWENTY QUESTIONS demonstrates a fundamental limitation on the power of any branching-search method. In the game one player thinks of an object, which he describes as being animal, vegetable or mineral, or any combination of those characteristics. The opposing players try to guess what the object is by asking as many as 20 questions, which must be answered "Yes" or "No." It can be proved that the players cannot identify more than 2^{23}, or 8,388,608, objects correctly. The reason is that the set of characteristics animal, vegeta-

Cryptanalysis). When these words are arranged alphabetically in the locations KEY[1], KEY[2], KEY[3], . . . ,KEY[31] of a table, Algorithm B first compares the desired word x to the midpoint KEY[16], which is the word "I". If x is alphabetically less than "I", the next comparison will be with KEY[8], which is the word "by"; if x is greater than "I", the next comparison will be with KEY[24], which is "that". In other words, Algorithm B acts on the table of words by following a structure that looks like an upside-down tree, starting at the top and going down to the left when x is less and down to the right when x is greater [*see top illustration on page 72*]. It is not hard to see that any algorithm designed to search an ordered table purely by making comparisons can be described by a similar binary tree.

The tree for binary search is defined implicitly in Algorithm B by arithmetic operations on l, r and j. It can also be defined explicitly by storing the tree information in the table of words itself. For this purpose let LEFT[j] be the position in the table at which we are to look if word x is less than KEY[j], and let RIGHT[j] be the position at which we are to look if x is greater than KEY[j]. For example, binary search in a table of 31 words would have LEFT[16] equal to 8 and RIGHT[16] equal to 24, since the search starts at KEY[16] and then proceeds to either KEY[8] or KEY[24]. If the search is to terminate unsuccessfully after determining that the desired word x is less than KEY[j] or greater than KEY[j], we respectively let LEFT[j] equal 0 or RIGHT[j] equal 0. In the illustrations on page 72 those 0's are represented by little square nodes at the bottom of the tree.

The location of the first key to be examined in a binary tree is traditionally known as the root; in the 31-word example the root is 16. It is possible to construct search algorithms that do not start by looking at KEY[16], and these may well be more efficient than Algorithm B if some words are looked up much more often than others. A generalized tree-search procedure follows:

```
Algorithm C; tree search.
C1. [Initialize.] Set $j$ equal to the location of the root of the binary search tree.
C2. [Unsuccessful?] If $j = 0$, output $j$ and terminate the algorithm.
C3. [Compare.] If $x = \text{KEY}[j]$, output $j$ and terminate the algorithm. If $x < \text{KEY}[j]$, set $j \leftarrow \text{LEFT}[j]$ and go back to step C2. If $x > \text{KEY}[j]$, set $j \leftarrow \text{RIGHT}[j]$ and go back to step C2.
```

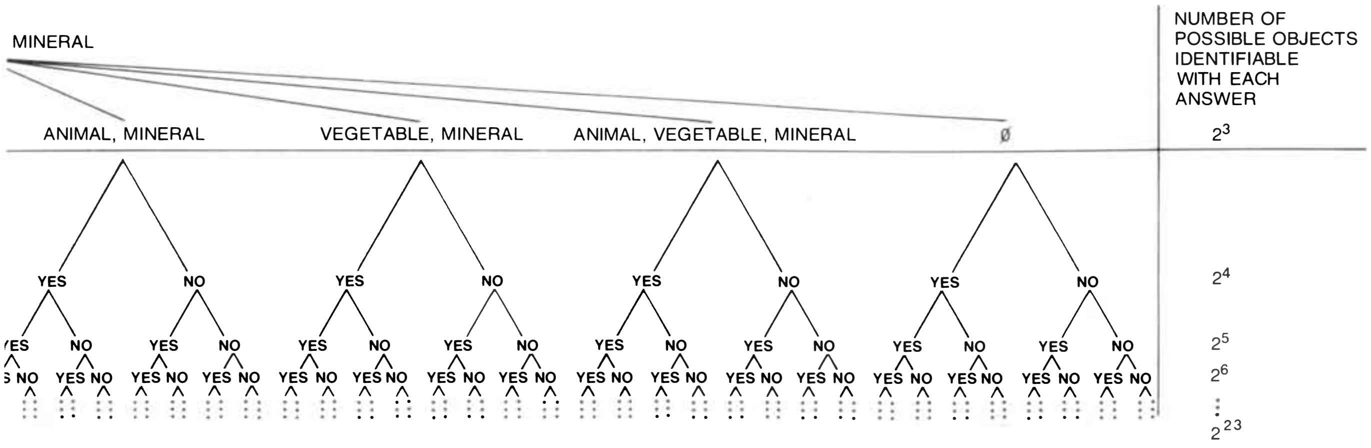
Algorithm C is analogous to a programmed textbook in which, depending on the answer to a certain question, each page tells the reader what page to turn to next. It works on any binary tree where all keys accessible from LEFT[j] are less than KEY[j] and all keys accessible from RIGHT[j] are greater than KEY[j], for all locations j in the tree. Such a tree is called a binary search tree.

One of the advantages of Algorithm C over Algorithm B is that no arithmetic calculation is necessary, so that the search goes slightly faster on a computer. The main advantage of Algorithm C, however, is that the tree structure provides extra flexibility because the entries in the table can now be rearranged into any order. It is no longer necessary that KEY[1] be less than KEY[2] and so on up to KEY[n]. As long as the pointers LEFT and RIGHT define a valid binary search tree, the actual locations of the keys in the table are irrelevant. This means that one can add new entries to the table without moving all the other entries. For example, the word "has" could be added to the 31-word binary search tree simply by setting KEY[32] ← "has" and changing RIGHT[j] from 0 to 32, where j is the location of the key "had". One might think that such additions at the bottom of the tree would upset the balanced structure, but it can be shown mathematically that if new entries are added in random order, the result will almost surely be a reasonably well-balanced tree.

Optimum Binary Search Trees

Since Algorithm C applies to any binary search tree, one can hand-tailor the tree so that the most frequently examined keys are examined first. Such tailoring reduces the average time required for a computer to carry out the search, although it cannot reduce the worst-case time. The bottom illustration on page 72 shows the best possible binary search tree for the 31 commonest English words, based on Gaines's estimates of their frequency. The average number of comparisons needed to search for x in this optimum binary search tree is only 3.437, whereas the average number of comparisons needed in the balanced binary search tree is 4.393. It is worth noting that the optimum tree, which is based on the frequencies of the words, does not start by comparing x with the word "the". Even though "the" is by far the commonest English word, it comes so late in alphabetical order that it is too far from the middle of the list to serve as the optimum root.

From the standpoint of conventional mathematics it is trivial to find the optimum binary tree for any particular set of n words and frequencies because there are only finitely many search trees. In principle one merely has to list all the trees and choose the one that works best. In practice, however, this observation is useless because the number of possible binary trees with n elements is equal to $(2n)!/n!(n+1)!$, where $n!$ stands for the product $1 \times 2 \times 3 \times \cdots \times n$. This for-



MINERAL
ANIMAL, MINERAL
VEGETABLE, MINERAL
ANIMAL, VEGETABLE, MINERAL
∅
YES
NO
NUMBER OF POSSIBLE OBJECTS IDENTIFIABLE WITH EACH ANSWER
2^3
2^4
2^5
2^6
2^{23}

ble and mineral has only eight, or 2^3, possible subsets (including the null set ∅ for an object with none of those characteristics), and these eight possibilities combine with only 2^{20} possible outcomes to the 20 yes-no questions. A similar argument can be used to show that a search algorithm asking at most 20 "less-equal-greater" questions cannot distinguish more than $2^{20} - 1$ different key values, since $1 + 2 + 4 + 8 + \cdots + 2^{19} = 2^{20} - 1$. Binary search is able to attain this maximum limit, thus it is the most efficient search algorithm of its kind.

mula shows that there are very many binary trees indeed, approximately $4^n/\sqrt{(\pi n^3)}$ of them, where π is the familiar 3.14159. For example, when n is 31, the total number of possible binary trees is 14,544,636,039,226,909, and each of these 14 quadrillion trees will be optimum for some set of assumed frequencies for the 31 words. How, then, is it possible to show that the particular tree I have chosen is the best one for Gaines's frequencies? The fastest modern computer is far from fast enough to examine 14 quadrillion individual possibilities; if one tree were considered per microsecond, the task would take 460 years.

There is, however, an important principle that does make the computation feasible: Every subtree of an optimum tree must also be optimum. In the optimum binary search tree for the 31 commonest English words the subtree to the left of the word "of" must represent the best possible way to search for the 20 words "a", "and" and so on over to "not". If there were a better way, it would lead to a better overall tree, and the given tree would therefore not be optimum. Similarly, in that subtree the even smaller subtree to the right of "for" must represent the best possible way to search for the 11 words "from", "had" and so on over to "not". Each subtree corresponds to a set of consecutive words KEY[i], KEY[$i + 1$], . . . ,KEY[j], where $1 \leq i < j \leq n$. It is possible to determine all the optimum subtrees by finding the small ones first and doing the computation in order of increasing values of $j - i$. For each choice of indices i and j there are $j - i + 1$ possible roots of the subtree. As one proceeds up the tree with the computation and examines each possible subtree root the optimum subtrees to the left and right will have already been calculated.

By this procedure the best possible binary search tree for n keys and frequencies can actually be found by doing about n^3 operations. In fact, I have been able to improve the method even further, so that the number of operations required can be reduced to n^2. In the case of the 31 commonest words this means that the optimum binary search tree can be discovered after only 961 steps instead of 14 quadrillion.

I should point out that the preceding paragraphs discuss several algorithms whose sole purpose is to determine the best binary search tree. In other words, the output of those algorithms is itself an algorithm for solving another problem! This example helps to explain why computer science has been developing so rapidly as an independent discipline. In the study of how to use computers properly, problems arise that are interesting in their own right, and many of these problems require both a new and interrelated set of concepts and techniques.

It is amusing and instructive to consider the worst possible binary search tree for the 31 commonest English words in order to see how bad things could possibly become with Algorithm C. As in the case of the optimal trees, there is a way to determine such "pessimal" trees in about n^2 operations. For the 31 words with Gaines's frequencies the pessimal binary search tree requires Algorithm C to make an average of 19.158 comparisons per search. By way of comparison the worst arrangement of the keys for a sequential search requires Algorithm A to make an average of 22.907 comparisons per search. Hence even the worst case for Algorithm C can never be quite as bad as the worst case for Algorithm A.

Hashing

The above algorithms for searching are closely related to the way people look for words in a dictionary. There is actually a much better way to search through a large collection of words by computer. It is called hashing, and it is a completely different approach that is quite unsuitable for human use because it is based on a machine's ability to do arithmetic at high speeds. The idea is to treat the letters of words as if they were numbers ($a = 1$, $b = 2$, $c = 3$ and so on through $z = 26$) and then to hash, or scramble, the numbers in some way in order to get a single number for each word. The number is the "hash address" of the word; it tells the computer where to look for the word in the table.

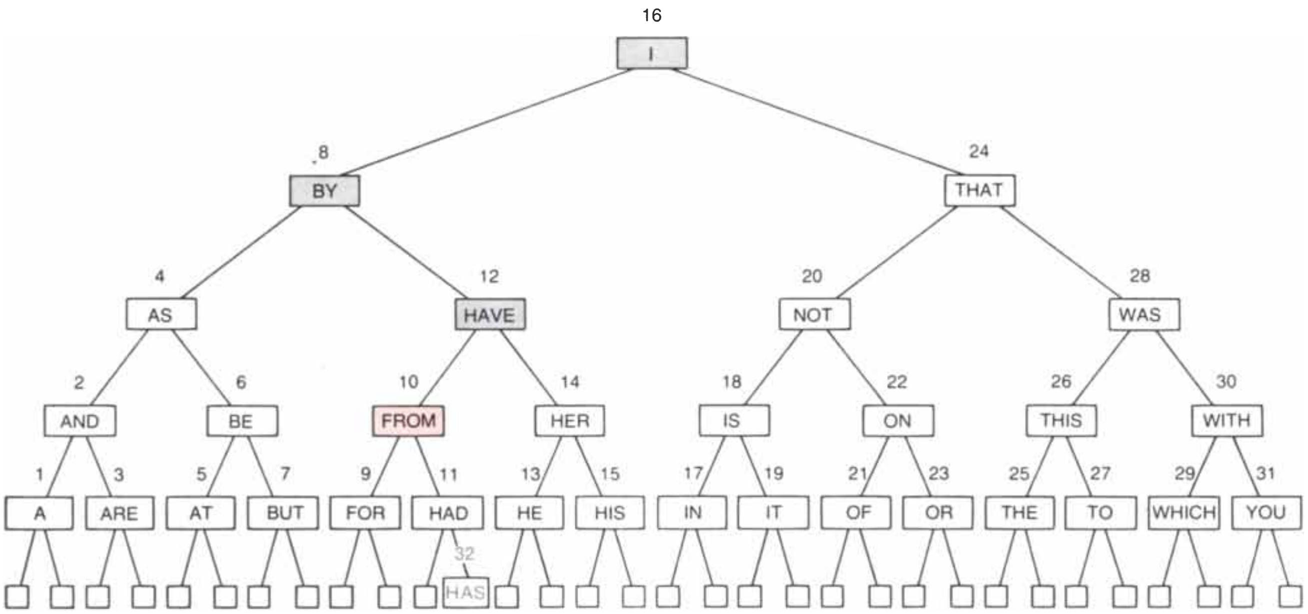
In the case of the 31 commonest English words we could convert each key into a number between 1 and 32 by adding up the numerical values of its letters and throwing away excess multiples of 32. For example, the hash address of "the" would be $20 + 8 + 5 - 32 = 1$, the hash address of "of" would be $15 + 6 = 21$, and so on for the rest of the list. If one is lucky, each word will lead to a different hash address and any search will be very fast.

In general, suppose there are m locations in the computer's memory, and suppose we want to store n keys, where m is greater than n. Since n is equal to 31, let us say m is equal to 32. Suppose also there is a hash function $h(x)$ that converts each possible word x into a number between 1 and m. A good hash function will have the property that $h(x)$ is unlikely to be equal to $h(y)$, if x and y are different words to be put into the table.

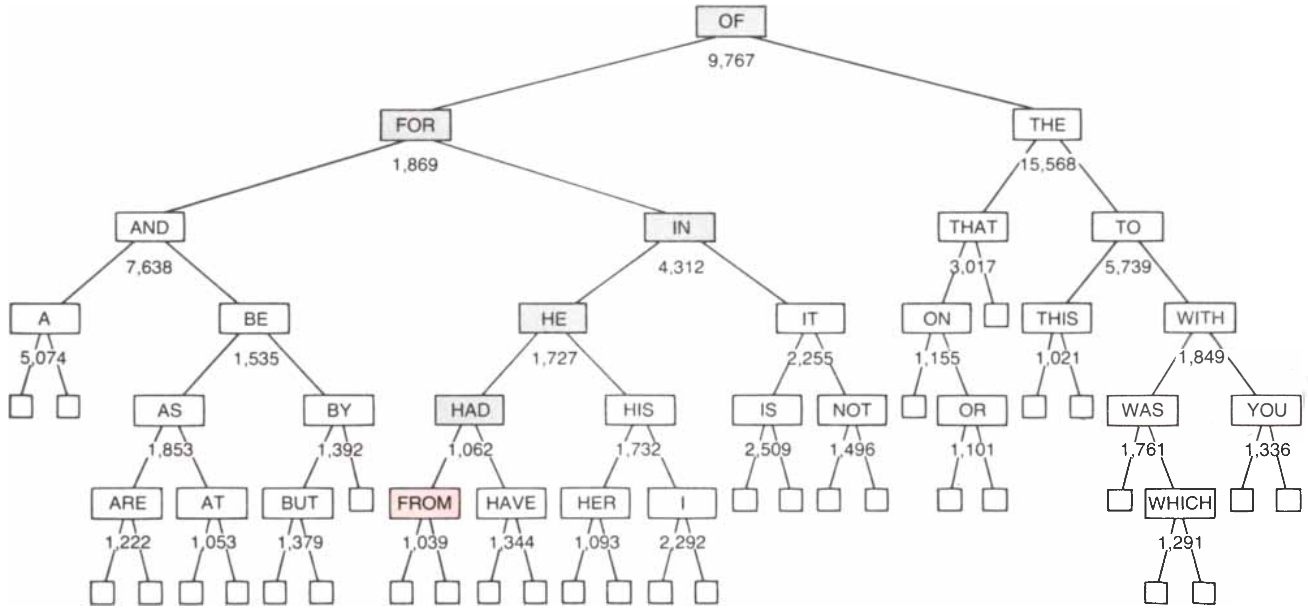
Unless m is much larger than n, however, nearly every hash function will lead to at least a few "collisions" between the values $h(x)$ and $h(y)$. It is extremely improbable that n independent random numbers between 1 and m will all be different. Consider a common example: It is well known that when 23 or more people are present in the same room there is a better than even chance that two of them will have the same birthday. Moreover, in a group of 88 people it is likely that there will be three individuals with the same birthday. Although this phenomenon seems paradoxical to many people, the mathematics can be easily checked, and many seemingly impossible coincidences can be explained in the same way.

Another way to state the birthday paradox is to say that a hash function with m equal to 365 and n equal to 23 will have at least one collision, more often than not. Thus any search procedure based on a hash function must be able to deal with the problem of collisions.

Suppose we want to search a table for x but the hash address $h(x)$ already contains word y. The simplest way to handle the collision is to search through locations $h(x)$, $h(x) - 1$, $h(x) - 2$ and so on until we either find x or come to an empty position. If the search runs off one end



BINARY SEARCH TREE is implicit in Algorithm B. Here a tree graphically illustrates the order in which Algorithm B would probe an alphabetical table of the 31 commonest words in English. Starting at the "root," or top, of the tree, the input word x is first compared with the midpoint of the table, the word "I". If x is alphabetically smaller than "I", the search proceeds down the left branch of the tree; if x is greater, the search proceeds down the right branch. For example, if x is the word "from", the search first finds that x is less than "I", then that x is greater than "by", then that x is less than "have", finally that x is equal to "from". If x were not in the table, the search would stop at one of the 32 zeros (*square nodes*) at bottom of the tree. When branches of tree are represented explicitly in computer's memory, rather than implicitly as in Algorithm B (which requires calculation of midpoints), search goes slightly faster. It also becomes easier to insert new information: if one wants to add "has" (*word in gray*) to tree, one inserts it in alphabetical order in place of one of zeros.



OPTIMUM BINARY SEARCH TREE shows the best order of the 31 words in the tree, based on the relative frequency of each word estimated by Helen Fouché Gaines. The frequency of each word is represented by the number below it. This tree is not as well balanced as the tree implicitly defined by the standard binary-search algorithm and shown in illustration above, and the search will therefore take longer in some cases. For example, to find the word "from" in this tree takes six steps instead of four (*path in gray and color*). On the average the optimum tree is faster for a computer to search, however, because the commoner words are tested sooner. Note that although the word "the" is by far the most frequently used word in English, it is not placed at root of the tree because it is too far from center of alphabet.

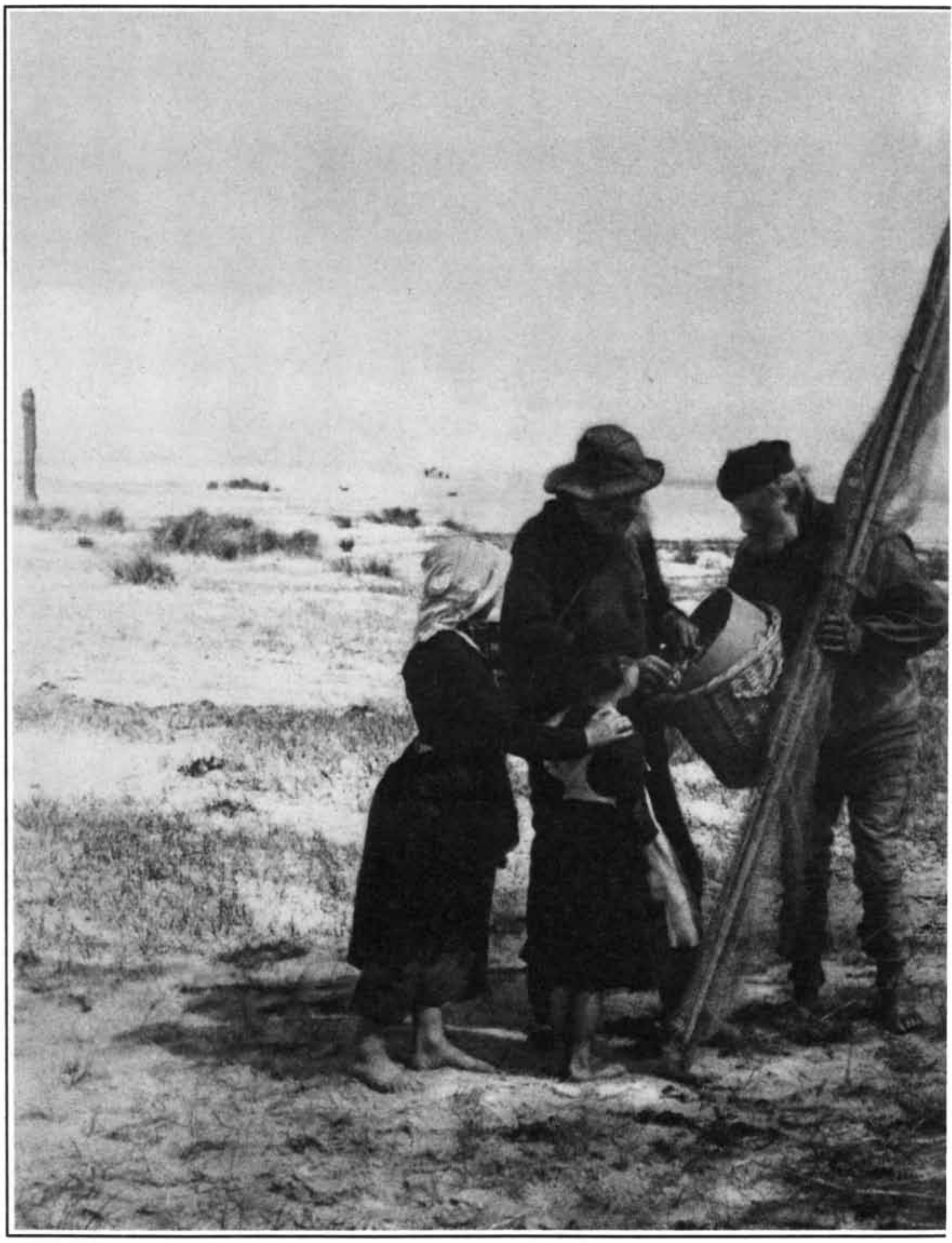
Grapes, like children, need love and affection.

Here in the Almadén Vineyards in Northern California we coddle and protect our children, the grapes of Almadén.
The fruit of our efforts can be seen in our versatile Grenache Rosé.
Carefully nurtured from the descendants of the famed Grenache grapes of Tavel, this Rosé is fruity, light and refreshing.
It's the perfect "goes with everything" wine.
Yes, we are proud parents.

Almadén Vineyards, Los Gatos and Paicines, California

Almadén

Polaroid Type 665 Positive/Negative Film is now a preservative agent: The Royal Photographic Society of Great Britain chose it to copy over 12,000 rare masterpieces (like the one below) from its Archives. Type 665 provides instant 3¼ x 4¼ inch positives for reference purposes. And negatives of superb quality for high fidelity reproductions. The negatives' high resolution permits great enlargement without loss of detail. And their wide tonal range reproduces every shading nuance of the orig-



FROM THE ARCHIVES OF THE ROYAL PHOTOGRAPHIC SOCIETY OF GREAT BRITAIN.

THE SI

(1887, 18½"× 11

Photographed by

inal prints. Furthermore, since the Polaroid prints and negatives are produced in only 30 seconds, the original photographs can lie, untouched, under the camera until the archivist knows he has satisfactory copies. The irreplaceable originals can then be removed to safe storage under optimum conditions with no further handling risks. Thus, in a manner of speaking, some of the great pioneers in the history of British photography will owe their lasting reputations to a film made by Polaroid Corporation.

REPRODUCED FROM A POLAROID TYPE 665 NEGATIVE (3¼ X 4¼)

IMPERS.

ATINUM PRINT)

nry Peach Robinson

The new $2984* Colt. Isn't a Datsun. Isn't a Toyota. It's a lot of little Dodge.

The new Dodge Colt is such a lot of car, it's got Mr. D. and Mr. T. confused. Because Colt offers you the value you'd expect from this import, plus Dodge Colt sales and service coast to coast.

47 MPG highway, 30 MPG city. Colt will give you great mileage, according to EPA estimates.** Your mileage may vary, according to your car's condition, equipment, and your driving habits. And Dodge Colt runs on either regular or unleaded gas.

Looking for a long list of standard features? Well, you get it on all of our '77 Colt models. Even our lowest priced two-door coupe gives you whitewall tires, two reclining bucket seats, tinted glass in all windows, carpeting, adjustable steering column, simulated wood-grained instrument panel, four-speed manual transmission, quiet sound insulation, trip odometer, locking gas cap, and electric rear window defroster.

And we offer you an optional automatic transmission to go with the standard 1.6 liter engine.

So if you're thinking "import," think about Dodge Colt. It's not a Datsun. Not a Toyota. For $2984, it's a lot of little Dodge. See it at your Dodge Colt Dealer's.

*Manufacturer's suggested retail price, not including destination charge, taxes, title, and options. California prices higher.
**Equipped with standard 1.6 liter engine, four-speed manual transmission, and 3.31 rear axle ratio. California mileage lower.

"Very nice, Mr. D."
"I thought it was yours, Mr. T"
COLT
Dodge
A PRODUCT OF CHRYSLER CORPORATION

Dodge Colt. It's a lot of little Dodge.

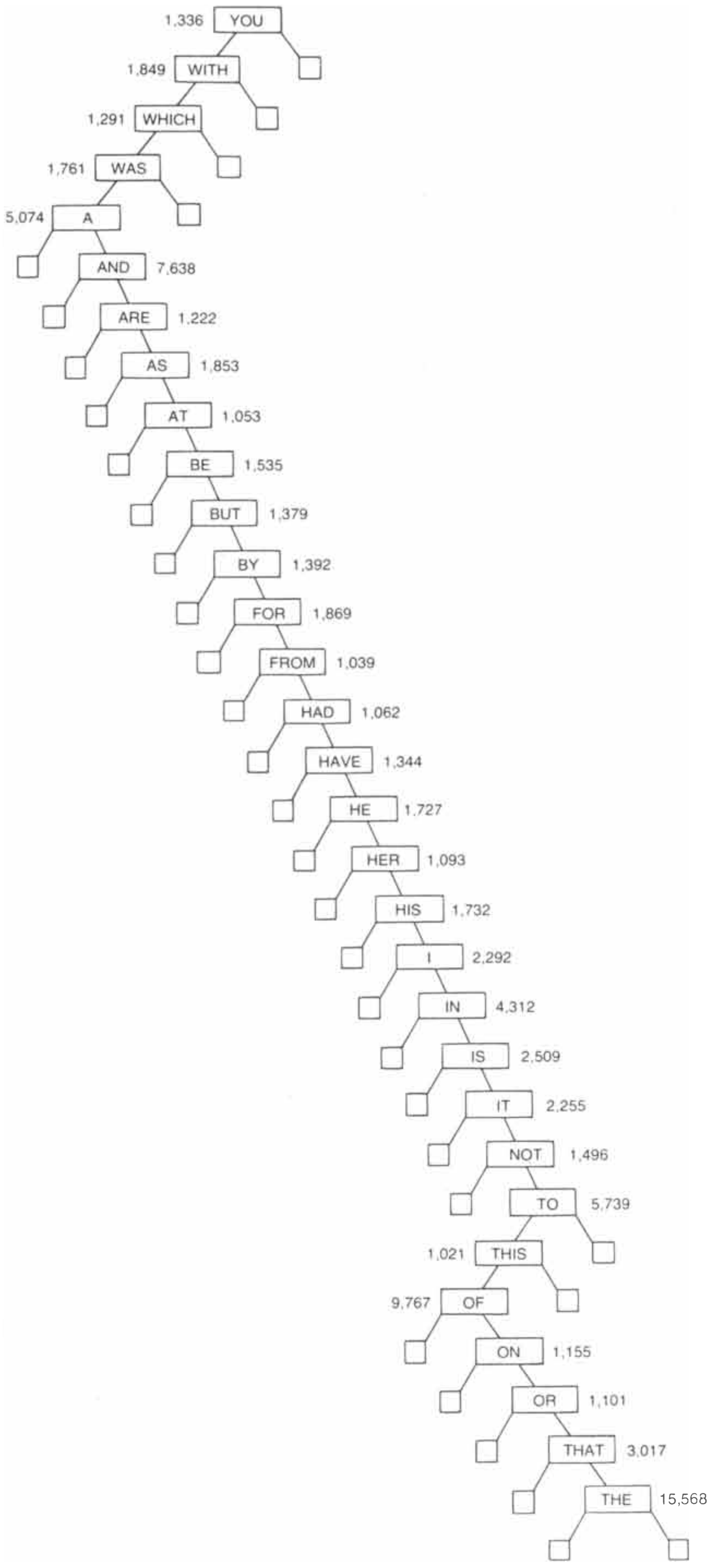
of the table before it is completed, it resumes at the other end. This procedure, which is known as linear probing, can be spelled out in an algorithm:

```
Algorithm D; hashing with linear probing.
  D1. [Initialize.] Set $j \leftarrow h(x)$.
  D2. [Unsuccessful?] If table entry $j$ is empty, output 0 and terminate the algorithm.
  D3. [Successful?] If $x = \mathrm{KEY}[j]$, output $j$ and terminate the algorithm.
  D4. [Move to next location.] Set $j \leftarrow j - 1$; then if $j = 0$, set $j \leftarrow m$. (Location $m$ is considered to be next to location 1.) Return to step D2.
```

If x is not in the table, and if the algorithm terminates unsuccessfully in step D2 because table entry j is empty, we could set $\mathrm{KEY}[j] \leftarrow x$ using the current value of j. This would insert x into the table so that the algorithm could retrieve it later. A subsequent search for x will follow the same path as it did the first time, starting at position $h(x)$, moving to $h(x) - 1$ and so on, finding x in position j. Thus the search will proceed properly even when collisions occur.

Returning to the example of the 31 commonest English words, suppose the words are inserted one by one into an initially empty table in decreasing order of their frequency ("the" is inserted first, "of" is inserted second and so on). The result is the hash table shown in the illustration on the next page. Most of the words appear at or near their hash addresses except for the ones that are inserted into the table last; the least frequent word, "this", has been placed in position 8 although its hash address is 24, because positions 9 through 24 were already filled by the time it was inserted. In spite of such anomalies the average number of times the table must be probed by Algorithm D in order to find a word turns out to be only 1.666—less than half the average number of comparisons required to find the word with the optimum binary search tree. Of course, the time needed to compute $h(x)$ in step D1 must be added to the time for probing the table. For large collections of data, however, the hashing method will significantly outperform any binary-comparison algorithm.

In practice one would almost never let a hash table get as full as it does in the example. The number m of table positions available is usually chosen to be so large that the table will never become more than 80 or 90 percent full. It can be shown that the average number of



"PESSIMAL" TREE shows the worst possible binary search tree for searching for the 31 commonest English words. This tree has lost advantage of tree structure because one branch of each comparison is always "dead."

1	THE	(1)
2	HAVE	(4)
3	TO	(3)
4	HIS	(4)
5		
6	BE	(7)
7	FOR	(7)
8	THIS	(24)
9	I	(9)
10	BUT	(11)
11	WAS	(11)
12	HAD	(13)
13	HE	(13)
14	FROM	(20)
15	AT	(21)
16	NOT	(17)
17	THAT	(17)
18	WHICH	(19)
19	AND	(19)
20	AS	(20)
21	OF	(21)
22	ON	(29)
23	IN	(23)
24	ARE	(24)
25	YOU	(29)
26	BY	(27)
27	WITH	(28)
28	IS	(28)
29	IT	(29)
30	HER	(31)
31	OR	(1)
32	A	(1)

"HASH" TABLE provides a better way for a computer to search through large files of data. For each word x one uses a computer's ability to do high-speed arithmetic by computing a hash address $h(x)$, where the search for x is to start. The hash address for each of the 31 commonest words is shown in parentheses after each word; in this example each hash address was obtained by adding the numerical value of each letter ($a = 1$, $b = 2$ and so on up to $z = 26$) and discarding excess multiples of 32. Sometimes two different words x and y have the same hash address $h(x)$, so that they "collide." If x is not stored in position $h(x)$, the search continues upward through positions $h(x) - 1$, $h(x) - 2$ and so on. For example, the hash address of "his" is $h + i + s$, or $8 + 9 + 19 - 32 = 4$. The hash address of "have" is also 4. To search for "have" the algorithm looks first in position 4 (*light gray*), then in position 3 (*dark gray*) and finally in position 2 (*color*), where "have" is located. If word x is not in table, search for it will stop at empty position 5.

probes needed to find one word out of n equally likely words that have been randomly inserted into a table of size m is $1 + [(n-1)/m + (n-1)(n-2)/m^2 + (n-1)(n-2)(n-3)/m^3 + \cdots]/2$. Let the symbol α stand for n/m, the fullness ratio or "load factor" of the table. As n approaches infinity it can be shown that the average number of probes required to find any word x in a table approaches the value $1 + (\alpha + \alpha^2 + \alpha^3 + \cdots)/2$, which is equal to $[1 + 1/(1-\alpha)]/2$. Furthermore, the true average number of probes will always be less than this limiting value. Therefore when the table being searched is 80 percent full, Algorithm D makes fewer than three probes per successful search on the average.

It is important to note that the stated upper limit on the average number of probes per successful search holds for all tables that are equally full, no matter how large the table is. The same cannot be said about binary-comparison algorithms, because their average running time per successful search will grow arbitrarily large as the number n of words to be searched increases.

Improving Unsuccessful Searches

My statements in the preceding paragraphs about the small number of probes required with Algorithm D apply only to cases where x is actually found in the table. If x is not present, the average number of probes needed to ascertain that fact will be larger, namely $1 + [2n/m + 3n(n-1)/m^2 + 4n(n-1)(n-2)/m^3 + \cdots]/2$; when n is large, this number is approximately equal to $[1 + 1/(1-\alpha)^2]/2$. In other words, an average unsuccessful search in a large hash table that is 80 percent full requires nearly 13 probes. Moreover, in my example of the 31 words in 32 spaces, note that all unsuccessful searches must terminate at the single empty position 5 regardless of the location of the starting address $h(x)$. A precisely analogous situation occurred with the sequential search Algorithm A, where all unsuccessful searches end at position 0.

In 1973 O. Amble of the University of Oslo noted that the problem of unsuccessful search could be alleviated by combining the concept of hashing with the concept of alphabetical order. Suppose the 31 commonest English words are inserted into the table in decreasing alphabetical order instead of in decreasing order of frequency. Since the table is probed by starting at the address $h(x)$ and moving to $h(x) - 1$ and so on, all words lying between the address $h(x)$ and the actual location of x must be alphabetically greater than x lest there be a collision. A search for x can therefore be terminated unsuccessfully whenever a word alphabetically less than x is encountered. In other words, the following algorithm can be used:

Algorithm E; linear probing in an ordered hash table. This algorithm assumes that KEY[j] is 0 when entry j is empty, and that all words x have a numerical value that is greater than 0.

E1. [Initialize.] Set $j \leftarrow h(x)$.

E2. [Unsuccessful?] If KEY[j] $< x$, output 0 and terminate the algorithm.

E3. [Successful?] If KEY[j] $= x$, output j and terminate the algorithm.

E4. [Move to next.] Set $j \leftarrow j - 1$; then if $j = 0$, set $j \leftarrow m$. Return to step E2.

The advantage of Algorithm E is illustrated in the ordered hash table on page 80. Suppose one wants to determine if "has" is one of the 31 commonest English words. Its hash address is $8 + 1 + 19 = 28$. With Algorithm E the search terminates in six steps when it reaches $j = 22$ ("by") instead of continuing through the table until it encounters the empty table entry at $j = 5$.

In an ordered hash table the average number of probes per unsuccessful search is reduced to $1 + [n/m + n(n-1)/m^2 + n(n-1)(n-2)/m^3 + \cdots]/2$, and this number is always less than $[1 + 1/(1-\alpha)]/2$. Thus the limit for a successful search and the limit for an unsuccessful one are identical. On the average, when an ordered hash table is 80 percent full, Algorithm E will make less than three probes regardless of the size of n.

This is all very well if the ordered hash table has been set up by inserting the keys in decreasing alphabetical order as I have described it. In practice, however, one cannot always assume that the words of a table have been entered in such a manner. Tables often grow dynamically with use, and new words enter in random order. Although the structure of a binary tree (Algorithm C) and of an unordered hash table (Algorithm D) will handle dynamic growth with ease, the structure of an ordered hash table (Algorithm E) is not so obviously adaptable. Fortunately there is a very simple algorithm for inserting a new word into an ordered hash table:

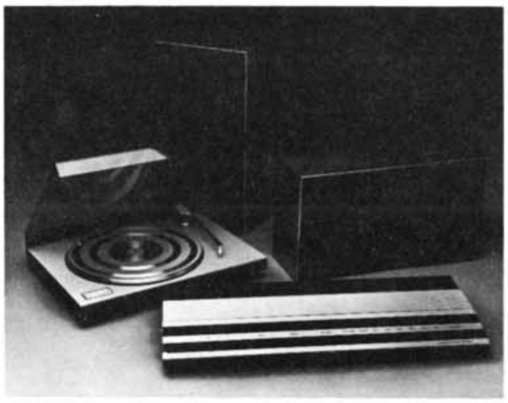
Algorithm F; insertion into an ordered hash table. This algorithm puts a new word x into an ordered hash table and appropriately rearranges the other entries so that searching with Algorithm E remains valid.

F1. [Initialize.] Set $j \leftarrow h(x)$.

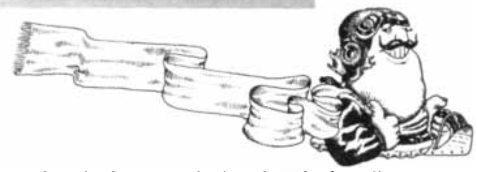
F2. [Compare.] If KEY[j] $< x$, interchange the values of KEY[j] and x. (That is, set x to the former value of KEY[j] and set KEY[j] to the former value of x.)

F3. [Done?] If $x = 0$, terminate the algorithm.

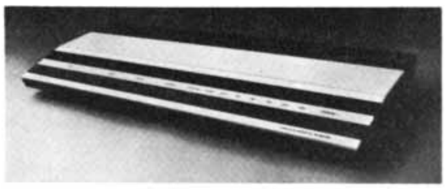
F4. [Move to next.] Set $j \leftarrow j - 1$; then



The Bang & Olufsen Beosystem 1900. It's so simple, most people don't get it.

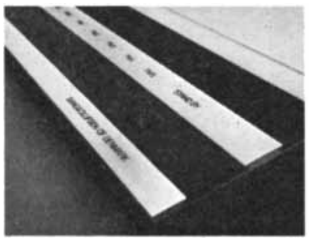


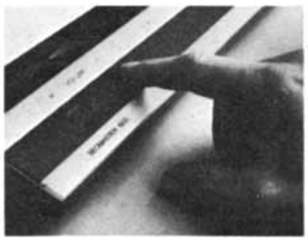
An alternative to the airplane cockpit school of audio design. It's not knobs and dials that make superb sound, it's superb engineering. In the Beosystem 1900, there is almost none of the former, and a great deal of the latter.



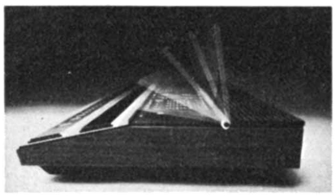
A few cases in point:

Your finger, the component. With the Beomaster® 1900, you become part of the system. All major controls are electronically activated by a light touch of your finger on the front control panel. The instant you touch it, an illuminated indicator appears for each function, you always know the operational status of the 1900, even in the dark.

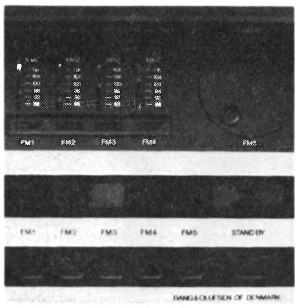




For details, look inside. Secondary controls, for bass, treble, and FM tuning, are out of sight, literally—concealed behind an aluminum door that opens and closes in a manner reminiscent of the Starship "Enterprise."



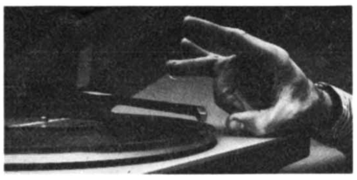
Thanks for the memory. The Beomaster 1900 also allows you one unforgettable convenience. You may pre-set the volume level and pre-tune up to five FM stations. Then, at the instant you want it, you have the station you want, at the level you want. Why clutter your memory when the system has one?

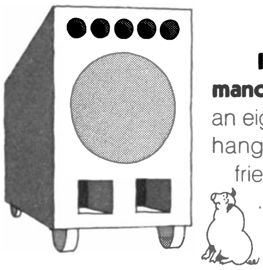


The turntable, taken to its logical conclusion. The Beogram® 1900 turntable's very low mass tone arm and MMC 4000 cartridge work magnificently with each other, because they are *made* to work with each other, by engineers who talk to each other, listen to each other, and design for each other. If that strikes you as overwhelmingly logical, you'd be surprised how other turntables are put together.

A scratched record is forever. (How to protect your investment.) No matter how little you've spent on your record collection, chances are some of it is irreplaceable, which makes it priceless. It makes sense to protect it—the way our MMC 4000 cartridge does with an effective tip mass of only 0.4 milligrams. (A tiny square of this page, this big □ weighs 1.0 milligram.) This results in a touch so delicate that it's almost impossible to scratch your records while playing them. It also reduces wear considerably enabling your records to continue working well past normal retirement age.

We don't recommend this, but with the MMC 4000 cartridge, it won't hurt.





It's not size that counts. It's performance. Can a speaker small enough to fit on an eight-inch shelf (or unobtrusive enough to hang on a wall) impress your audiophile friends? Yes, if they keep their eyes closed ...and their ears open.

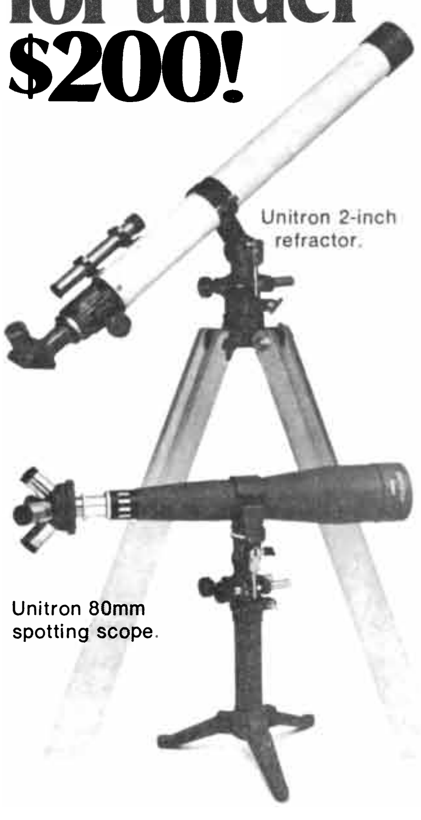
The missing link: Our 100% solution. Phase distortion—a principal villain in speaker performance—was identified in 1973 by Bang & Olufsen engineers. The first practical solution to the problem was presented in London in 1975 to the international organization, the Audio Engineering Society, by Bang & Olufsen engineers.

Today that solution is an integral part of our Beovox® Phase-Link® Loudspeakers (Pat. Pend.).

If a child can operate it, will an adult buy it? Because *usability* is at the heart of the Beosystem 1900's design, it is true that a child can operate it. But only a very sophisticated adult can truly appreciate it. Welcome. Write to us at: Bang & Olufsen of America, 515 Busse Road, Dept. 13K Elk Grove Village, Illinois 60007, we'll be happy to send you our brochure and dealer list.

Bang & Olufsen

Genuine Unitron telescopes for under $200!



Explore the moons of Jupiter or look at a squirrel through the extraordinary optics of a genuine Unitron telescope! The two-inch Unitron refractor for viewing the heavens is only $195 complete, and the new 80mm Unitron widefield spotting scope for terrestrial observations starts at *only* $90!

A nice hobby gift for young and old!

Send coupon for additional information and free catalog.

UNITRON The value line

Unitron Instruments, Inc.
Sub. of Ehrenreich Photo-Optical Industries, Inc.
101 Crossways Park West, Woodbury, NY 11797
USA (516) 364-8046 Dept. SA-4

Name____________________
please print
Address____________________
City____________________
State__________ Zip__________

if $j = 0$, again set $j \leftarrow m$. Go back to step F2.

If we choose to insert the word "has" into the ordered hash table of the 31 commonest English words by means of Algorithm F, the procedure would place "has" in position 22, making room for it by moving "by" from position 22 to position 18, moving "at" from position 18 to position 15, moving "as" from position 15 to position 14, moving "are" from position 14 to position 8 and finally moving "and" from position 8 to the empty position 5. That may seem like a lot of work, but it takes only slightly longer than the task of inserting "has" into an unordered hash table using Algorithm D. In general the insertion of a word into an ordered hash table takes the same number of iterations as the insertion of the same word into an unordered hash table. Furthermore, the average number of words in the table that must be interchanged by way of step F2 to accommodate the new word is $(n-1)/2m + 2(n-1)(n-2)/3m^2 + 3(n-1)(n-2)/4m^3 + \cdots$, which is approximately equal to $1/(1-\alpha) + [\log_e(1-\alpha)]/\alpha$, where e is the familiar 2.71828. Thus inserting words by Algorithm F is quite a reasonable task.

1	THE	(1)
2	HAVE	(4)
3	TO	(3)
4	HIS	(4)
5		
6	BE	(7)
7	FOR	(7)
8	AND	(19)
9	I	(9)
10	BUT	(11)
11	WAS	(11)
12	HAD	(13)
13	HE	(13)
14	ARE	(24)
15	AS	(20)
16	NOT	(17)
17	THAT	(17)
18	AT	(21)
19	WHICH	(19)
20	FROM	(20)
21	OF	(21)
22	BY	(27)
23	IN	(23)
24	THIS	(24)
25	IS	(28)
26	IT	(29)
27	ON	(29)
28	WITH	(28)
29	YOU	(29)
30	A	(1)
31	HER	(31)
32	OR	(1)

ORDERED HASH TABLE, which combines the concept of hashing with the advantage of alphabetical order, reveals more quickly when the input word is not present. Here all the words between position $h(x)$ and the actual location of x are alphabetically greater than x. Thus an unsuccessful search need not stop only at the empty position 5; it will also stop as soon as a word alphabetically less than x is encountered. If desired word x is "has", with hash address 28 (*light gray*), search will stop when it reaches "by" at position 22 (*dark gray*).

In this specific case we actually should not have inserted "has" into the table because hash tables ought to have at least one empty position. By coincidence the smallest possible word in alphabetical order, "a", is present in this completely full table. Hence linear probing with Algorithm E will still work correctly in all cases. If "a" were not in the table, however, an empty position would be needed in order to avoid endless searching when the input word x was equal to "a".

One of the most surprising properties of ordered hash tables is that each one is unique. If we use Algorithm F to build an ordered hash table from any set of words, the same table will be obtained regardless of the order in which the words are inserted. The reader may find it entertaining to prove this fact for himself.

Conclusion

My discussion of ways to search for information stored in a computer's memory is intended to illustrate several important points about algorithms in general. As we have seen, an algorithm must be stated precisely, and it is not as easy to do that as one might think. When one tries to solve a problem by computer, the first algorithm that comes to mind can usually be greatly improved. Data structures such as binary trees are important tools for the construction of efficient algorithms. When one starts to investigate how fast an algorithm is, or when one attempts to find the best possible algorithm for a specific application, interesting issues arise and one often finds that the questions have subtle answers. Even the "best possible" algorithm can sometimes be improved if we change the ground rules. Since computers "think" differently from people, methods that work well for the human mind are not necessarily the most efficient when they are transferred to a machine.

200-SX

Suddenly from Datsun: A sporty car with everything but a sports car price.

Exit dull, sluggish economy cars.
Enter Datsun's spicy 200-SX. Sweet-handling. Tasty appointments. And no bitter price to swallow. Enjoy.

Fun and frugal 5-speed.
Sporty 5-speed transmission works like overdrive. Thus, saving gas.

According to EPA estimates, 200-SX squeezes 34 MPG on the Highway, 23 City. Naturally, your actual mileage depends on driving habits, optional equipment and condition of car. California mileage lower.

Extras, yes. Extra cost, no.
- AM/FM multiplex stereo radio
- Steel belted radial tires
- Tachometer
- Fully reclining bucket seats
- Cut-pile carpeting
- Electric rear window defogger
- Tinted glass
- Electric clock
- Sporty 5-speed gearbox
- Power-assist front disc brakes

All for $4399! (Manufacturer's Suggested Retail Price not including destination charges, taxes, license or title fees and optional tape stripe and mag type wheel cover package.)

Tough sport.
Solid, all-steel unibody is but one example of how the Datsun 200-SX is put together to stay together. Fact is, when we made this fun little car, we made sure of one thing.
The fun would last.

Suddenly it's going to dawn on you.

DATSUN SAVES