Algorithms, Processes, and Machines

Peter J. Denning
DRAFT 1/28/01

Less than a generation ago, mathematicians and technologists were the only serious users of the word algorithm. Algorithms are now part of everyday conversation in many fields including engineering, science, literature, art, and music. They were even the subject of a political jibe: "What did Bill Clinton play on his sax? Al Gore rhythms."

Most people understand an algorithm as a step-by-step procedure, a recipe, a set of instructions telling how to get a task done. This simple definition has amazing consequences: algorithms can differ in speed by astronomical ratios, many problems have no fast solution algorithms at all, and some problems have no solution algorithm at all. Computers are universal simulators: any computer can *become* any other machine simply by loading into it an algorithm describing that machine. No other machines are of such kind. Some algorithms can produce exact copies of themselves, a phenomenon called *artificial life* -- but this sheds no light on whether algorithms can be intelligent or aware.

Algorithms have become not only ways to talk about what computers do, they have become a way of thinking. The algorithmic mind interprets worldly phenomena as manifestations of information and seeks to find algorithmic solutions for problems, even including those of the human condition. We describe our nervous systems as wiring, our habits as software, and our brains as computers made of meat. We think that education is a process of transferring information from teacher to student and learning as a process of acquiring and applying information. We worry that ultimately computers will be able to simulate human beings and that robots will replace humans as a new, more powerful species. This situation is little different from earlier eras -- we have a natural human tendency to identify with the current technology. For fifty years people have identified with their cars. In the nineteenth century, industrial metaphors were popular -- e.g., Taylor's scientific management, Freud's subconscious exerting pressures that could cause blowouts, Newton's clockwork universe, and generic phrases such as the machinery of government. But, as you will see in what follows, computing machines have such strong limitations that there remain many things that human can do much, much better. We have spent many years trying to make computer more like brains and all we have to show for it is brains that think they are computers.

What are Algorithms?

Algorithms as procedures for carrying out certain calculations have been known to mathematicians for over a thousand years. One of the earliest known works featuring formal methods was a book on logic and arithmetical procedures by Al-Khowarizmi in Arabia around AD 825. Both "algorithm" and "algebra"

derive from his name. Various famous algorithms acquired names down through the centuries, such as Euclid's algorithm for the greatest common divisor or Gauss's algorithm for solving a system of linear equations. The calculus of Newton and Leibniz in the 1700s yielded numerous algorithms for calculating functions of continuous physical processes such as planetary motions or steam engines. Algorithms for calculating tables of arithmetic functions (e.g., logarithms) were used by astronomers in the 1800s. Charles Babbage built the difference engine to calculate error-free numerical tables rapidly in the 1830s. Countess Ada Lovelace described algorithms for using Babbage's analytical engine, which he designed in the 1850s to calculate more general mathematical functions. Since the 1830s, logicians were interested in a universal algorithm (method) for proving theorems, a hope that was dashed for logic in 1930 by Gödel and for mathematics by Turing in 1936.

Algorithms are often defined as procedures that can followed mechanically to yield a desired *result* (output) starting from a given *input*. A *procedure* is a specification of steps or instructions that someone can follow to carry out a task. Examples of procedures include filing a tax return, renewing a drivers license, hiring people into a firm, diagnosing an illness, assembling a cabinet from a kit, installing software, and baking a cake according to a Julia Child recipe. These common examples all exhibit steps that may require some degree of human judgment. What happens if we restrict to steps that require no judgment at all? Such steps can be performed mindlessly by a human, or mechanically by a machine. A procedure consisting only of simple, unambiguous, mechanical steps is what we call an algorithm. Many useful procedures contain steps that require human judgments that cannot be mechanized; such procedures do not qualify as algorithms.

Algorithms can be guaranteed to work as intended only when the input is "legal" -- it meets stated requirements. A human being can usually tell when an input is inappropriate for an algorithm, but a machine cannot. When designing an algorithm, therefore, it is wise to include steps to test the input for validity before undertaking the remaining steps.

Algorithms have three kinds of steps. (1) A simple action -- e.g., mix the ingredients. (2) A choice that depends on a condition -- e.g., if the taste is flat, add salt, otherwise do nothing. (3) An iteration that terminates when a condition is met -- e.g., stir until thick. The iteration step allows short algorithms to take a long time to execute. It allows some algorithms to execute forever if the termination condition is never satisfied, a situation known as *infinite loop*.

These simple requirements have four far-reaching implications. The first implication is one of the biggest surprises of the computing era: some problems cannot be solved by any algorithm at all. Among the many noncomputable problems are ones of great practical interest, such as deciding whether a computer program ever stops, whether it meets its specifications, or whether it contains a virus. The second implication seems less surprising: problems solvable by algorithms come in varying degrees of difficulty. Since the 1970s, we have learned much about how to measure the difficulty of problems. We also

learned an astonishing fact: thousands of common problems in business, manufacturing, goods distribution, aerospace, aircraft design, transportation, scientific data analysis, engineering, computer chip design, economics, meteorology, geology, oceanography, and many more fields, are intractable. That means that any algorithm capable of solving the problem will always require enormous amounts of storage and time to accomplish its task, well beyond our most generous budgets and forbearance. A third implication is that exceedingly simple machines can execute algorithms of arbitrary complexity; in fact, there is one simple, universal machine that can simulate any other. In practical terms, this means that one computer can become any other simply by loading the software description of the other's algorithm into it. A fourth consequence is that machines are capable of constructing other machines, including copies of themselves, in a form of artificial life. These claims are explored below.

Although many examples of algorithms are drawn from mathematics, algorithms exist in all domains of human endeavor. The procedures used by many bureaucratic organizations are algorithms for processing various requests, often so detailed that people of little skill can carry them out successfully. Assembly lines in manufacturing plants are run by machines and by workers following detailed procedures. Driving directions are usually given as algorithm to move from one location to another. Many parts of the US Tax Code are written as algorithms because the lawmakers could not state a general principle to be followed. Many training manuals lead students through algorithms to teach them how to do standard procedures.

```
INPUT: specified ingredients
1. Place ingredients in bowl
2. Mix thoroughly
3. Add flour until thick
4. Add salt to taste
OUTPUT: serves 4
```

```
INPUT: integers m, n > 0
1. Divide m by n; let r be
   the remainder.
2. If r=0, stop, OUTPUT n.
3. Set m to n, n to r, and
   go back to step 1.
```

The procedure on the left resembles a common cookbook recipe. Each step makes sense to an experienced cook but is subject to different interpretations by different cooks. The recipe is therefore not an algorithm. The procedure on the right computes the greatest common divisor of m and n (the largest integer that divides both numbers). Each step is unambiguous and can be performed by a machine (or carried out mindlessly by a clerk trained in arithmetic). It is called Euclid's Algorithm. Euclid's Algorithm contains an iteration where it repeats the steps until r=0. Will it always be true that eventually r=0?

It is very important to distinguish between a procedure and a process. When you execute a procedure by carrying out its steps, you are carrying out the *process* evoked by that procedure. A process is the sequence of actions taken dynamically by a processor when following a procedure. Although procedures are always finite, processes can be infinite if the procedure allows loops. It's possible that the stopping condition will never be satisfied and the process will never terminate. This can be a real problem. If you are monitoring a program that hasn't stopped after a reasonable time, should you conclude it is in an infinite loop? Or that the input data just require a lot of processing and the machine will be done soon? In general it's impossible to tell whether a procedure specifies a finite or infinite process. (We'll say more about this shortly.)

Data Representations

Every algorithm operates on data stored in a memory. A portion of the memory holds input data and another portion output data. The arrangement of the data in the memory can have a significant influence on the algorithm's speed. A simple example is the white pages of a phone book. The names of the subscribers are listed alphabetically along with their phone numbers. Looking up a phone number for a given name is fast and easy. But finding the name associated with a phone number may require a tedious search, one entry at a time. In a phone book of 1,000,000 entries, it would take approximately 20 probes (reading one entry) to locate a given name alphabetically and about 500,000 probes to locate a name given a phone number.[1]

This means that the algorithm designer must think carefully about the data structure as well as the way of sequencing the algorithm's steps. Some data structures yield long execution times for the question the algorithm is to answer. For this reason, the issue of data structure, sometimes also called "information representation," is very important in the design of programs. Niklaus Wirth, the designer of the language Pascal, summarized this as "programs = algorithms + data."

The Turing Machine Model

Much of our modern understanding of algorithms is rooted in the work of Alan Turing. In his 1936 paper, a study of the possibility of machines performing numeric calculations, he proposed the model now known as Turing machine. At that time, the notion of algorithm was still imprecise; there was no accepted

[1] For those interested in the details, they are as follows. Define a "probe" to mean looking at one entry in the table. If the table is alphabetical, we can use a "binary search" by probing the list at the middle and then comparing the name sought with the name probed to determine which half-list to search next. In a table of 2^n elements, the search would require n probes. A non-alphabetic, must be searched sequentially, which means that on average half the list (2^{n-1}) must be probed. In the example n=20 in a list of about 1 million elements.

definition. Turing set out to provide a mathematically rigorous definition and use it to answer questions about what mathematical functions could be computed by algorithms.

Turing's model of an algorithm is a machine consisting of a control unit and a tape comprising a sequence of cells. Each cell is either blank or holds a symbol from a predetermined (and finite) alphabet. At each moment, the control unit is positioned at one of the cells; it reads the symbol in the cell and then, based on its internal state, it writes a new symbol and moves one cell either left or right. The control unit is governed by an algorithm composed of these very simple steps. Turing argued that a mathematician writing out a proof or a derivation engages in such simple steps at successive, discrete moments in time. He concluded that this simple model could mimic any sequence of actions that a mathematician might use. The model was both simple and general.

(a) #X#
(b) #X# → #Y#
(c) #X# → #Y# → #Z#

The contents of a Turing Machine's tape can be represented as a string of letters X bounded by # symbols (a). The # symbols denote blank tape cells and are not used as part of the input and output strings. The transformation of an input tape to an output tape by a function Y=f(X) can be written with an arrow separating the input and output tapes (b). By convention, the machine starts positioned at the # symbol immediately to the left of the input and stops at the # symbol immediately to the left of its output. This convention enables algorithms for multiple functions to be cascaded together, as in (c).

```
1: R; a=cell; if a=# then goto 5; cell=$
2: {R} until cell=#
3: {R} until cell=#; cell=a
4: {L} until cell=$; cell=a; goto 1
5: {L} until cell=#; L; halt
```

A copy machine makes an identical copy of a string on the tape. The symbol # denotes a blank cell and $ is a special marker symbol that tells how much of the input has been copied. The initial configuration is #X# for some nonempty string of symbols X, with the machine positioned at the # just to the left of X. The final configuration is #X#X# with the machine positioned at the # to the left of the left X. The copy algorithm's strategy is to mark one letter of X with $, move to the right end of the tape and write a copy of that letter, and then return to the $ and change it back. Thus if X=UaV at an intermediate step, the machine's configuration after one of these cycles will be #U$V#Ua#. Turing machine algorithms, such as this one, use very simple instructions. Instruction R means move right one cell, L means move left. The instruction "a=cell" means read the letter of the current cell into a register named "a"; conversely, "cell=a" means to write register "a" to the current cell. Statement 2 means "keep moving right until a cell containing # is encountered." Statements 3-5 have similar meanings. Statement 5 positions the machine at its conventional stopping place, and then stops.

Turing argued that mathematical calculations can be decomposed into primitive steps -- e.g., copy, search for a symbol, write down symbols, or add two numbers -- capable of being executed by his machine model. He speculated that any procedure describable as a sequence of unambiguous steps could be formalized as a Turing machine. He argued that standard arithmetic and algebraic functions are effectively computable because we can easily specify algorithms for them. So is the process of verifying a mathematical proof given the axioms and the rules of inference.

The Church-Turing Thesis

During the 1930s and 1940s two other mathematicians investigated methods of expressing mathematical functions. Alonzo Church developed a notation that he called the lambda-calculus. He claimed that only the functions described by lambda-expressions could be computed and that the lambda-expression gives the computational directions. (In the late 1950s, John McCarthy invented a

programming language called LISP to automate the process of evaluating expressions in Church's calculus; he and many others used LISP in their explorations of intelligent machines.) Because both Church and Turing were making a similar claim, and because it was easy to build Turing machines to evaluate lambda-expressions and to write lambda-expressions that simulate Turing machines, the two claims were merged into one and were called the Church-Turing thesis: *Any procedure describable as a set of unambiguous, simple steps can be realized by an appropriate Turing machine.*

In the 1940s, the mathematician Stephen Kleene studied another system called recursive functions. These are functions whose higher values are expressed in terms of lower values; for example, the factorial f(n)=n! can be written as f(n) = n·f(n-1) with f(0)=1. Kleene showed that any function expressible in Church's calculus is also expressible as a recursive function and can be evaluated by a Turing machine. This added further weight to the Church-Turing claim. No one has ever found another system for expressing algorithmic problems that is more powerful than the Turing machine. Today, Turing machines serve as the reference model for anything claimed to be computable.

Noncomputable Functions and Undecidable Problems

The moment you propose that a Turing machine gives a precise way to compute a mathematical function, you open the question, "Are there functions that cannot be computed? Functions for which no algorithm exists?" Turing offered an astonishing answer.

Turing cited the halting problem as an example of a problem with no algorithmic solution. The halting problem is this: give a machine M (described by its program) and an input X, does M halt for X? Although at the time he studied this problem electronic computers were still on the drawing boards, Turing knew full well that programmers would confront the halting problem every day. Programmers (and, today, software engineers) are constantly concerned with whether they can prove their programs stop for every input with the correct answer. Turing's demonstration reverberates through time, a silent message to programmers in every age that they must deal with the halting question individually for every algorithm they construct.

Turing's argument about the impossibility of an algorithm for solving the halting problem is both simple and mind-boggling. Suppose that such a machine exists; let's call it H. When H is presented with the input #M#X#, it will eventually halt with the output #0# (meaning "M does not halt for X") or #1# (meaning "M halts for X"). Here M is a string specifying a Turing machine program. If H exists, we can easily build another machine K as follows. K is presented with the input #M#; it makes a copy of M using the copy routine and passes #M#M# to H. If H reaches its "M halts" state, K enters an infinite loop. If H reaches its "M does not halt" state, K halts with output #0#. What happens when K is applied to input #K#? If K halts for this input, then its embedded H must enter its "subject machine halts" state which, by construction, forces K into a loop. This is a contradiction: hence K must not halt for input K. But wait -- if K does not halt for input K, then the H embedded in K must enter its "subject machine does not halt" state which means that K prints #0# and halts. No matter what we assume -- K halts or K does not halt -- we have a contradiction. Since the construction of K is valid given the existence of H, the only conclusion is that H itself cannot exist.

In the years following Turing's discovery, many other problems were found to be noncomputable. These included: How long must one watch a machine executing a program before one can definitely conclude the program is in an infinite loop? How much storage will a program use? What is the longest output that a program of N instructions can print without going into an infinite loop? Does a given program meet its written specifications? In the middle 1980s, when the first computer viruses appeared, virus researchers discovered that there is no general method to tell if a program contains a virus. For if such a method exists, it could also be used to solve the halting problem. This means that the best that virus detectors can do is recognize particular viruses or particular actions known to be characteristic of viruses. But there is no one, fixed algorithm to detect viruses. (If there were, the antivirus software industry would not exist.)

Most of the proofs of impossibility of certain algorithms rely on self-reference. This means that if we have an algorithm that can answer a question about any machine, we can ask, what happens if we ask the algorithm about itself? If that produces a contradiction, the algorithm cannot exist. Sometimes proofs of this kind strike newcomers as logic puzzles -- oddities good as brain teasers but not obstacles to resolving practical questions. But nothing could be farther from the truth. Self-referential contradictions appear in virtually every language and no

one has ever found a way to eliminate them. A famous example is Russell's Paradox, a simple form of which is the statement, "I always lie." Self-referential paradoxes are part of the deep mysteries of language. They are mind-boggling, and they forever frustrate our attempts to answer certain questions. If this subject fascinates you, read Doug Hofstadter's Pulitzer-Prize winning book *Gödel, Escher, Bach: The Eternal Golden Braid* (Harper, 1984), where he explored at great depth the strange, self-referential loops in the logic of Kurt Gödel, the art of Martin Escher, and the music of Johann Sebastian Bach.

It might seem that we can make an algorithm that will generate all Turing machine programs. Such an algorithm would generate a sequence of programs, say M(1), M(2), M(3), ... Once we have any one of these programs, we can send it to a universal Turing machine for evaluation on any input. Let M(i,X) denote the output computed by M(i) for input X. For example, we can compute M(3,2), the result of applying program M(3) to input tape containing the number 2. Now let us construct another machine N such that N(i) = M(i,i)+1. The machine N uses the enumeration algorithm to find the program M(i), sends it to the universal machine with input i, and adds 1 to the result. The problem is that machine N cannot be enumerated by the algorithm! For if it were, N would be the same as M(k) for some k, but N(k) would then be N(k) = M(k,k)+1 = N(k)+1. This cannot be! The only conclusion is that, given *any* algorithm for enumerating Turing machines, there will be well defined machines that cannot be enumerated by that algorithm. This argument is the basis of Gödel's incompleteness theorem. It demonstrates why self-reference always causes so much trouble when we are trying to figure the limits of computation.

Degrees of Difficulty of Algorithmic Problems

Not only does the Turing machine model enable us to make distinctions between computable and non-computable problems, it allows us to make distinctions among computable algorithms according to the amount of computational work done by the algorithm. The model does this by giving us a means to measure the time and storage required by an algorithm. It is bad news that important questions cannot be computed by any algorithm. But it gets worse. Being computable is no guarantee that a problem can be solved within practical limits on time and storage.

The length of the input is a measure of the size of the problem an algorithm is asked to solve. Most algorithms take progressively longer to solve progressively bigger versions of the problem. We usually express the running time as an

order-of-magnitude estimate of the number of operations required to solve a problem of size n. Thus, if an algorithm performs $3n^2$ operations for input of size n, we would say that the running time is of order n^2. We use the "Big-Oh" notation $O(n^2)$ to mean "order of n^2". The box below illustrates with two sorting algorithms. The bubblesort takes time $O(n^2)$ and quicksort, which is the fastest sorting algorithm possible, takes time O(n log n).[‡]

```
procedure bubblesort(1,n)
for i=1 to n-1 do
  for j=i+1 to n do
    if a[j]<a[i]
      then swap(a[i],a[j])
  end
end
return
```

```
procedure quicksort(i,j)
  if i≥j return
  v=a[i]; p=i; q=j
  while p≤q do
    if a[p]≤v then p=p+1
    else if a[q]>v then q=q-1
    else swap(a[p],a[q])
  quicksort(i,p-1)
  quicksort(p,j)
return
```

These two sorting algorithms illustrate the effect of algorithm strategy on running time. Both algorithms take an input array of numbers a[1]...a[n] and rearrange them into ascending order. The bubblesort algorithm (left) iteratively works this strategy: assume that elements a[1]...a[i] are sorted; move the minimum among the remaining elements into position a[i+1]; now a[1]...a[i+1] are sorted. The running time of bubblesort is proportional to n^2. The quicksort algorithm (right) works this strategy: to sort a sublist a[i]...a[j], let v be the value of a[i]; swap elements so that a[i],...,a[p-1] are all less than or equal to v and a[p],...,a[j] are all greater than v; then sort the sublists a[i]...a[p-1] and a[p]...a[j]. This strategy is called recursive because the same strategy is used to sort smaller sublists. The entire list is sorted by quicksort(1,n). Quicksort takes time proportional to n log n. For large n, quicksort is significantly faster than bubblesort. No sorting algorithm is faster than quicksort.

"Order notation" makes it easier to compare algorithms without knowing the exact speed of the system in which is runs. For example, an $O(n^2)$ sorting algorithm that takes $3n^2$ operations would take about one hour on a 1950s

[‡] For those interested in the details, they are as follows. At the i-th pass, bubblesort examines the elements i+1,...,n. The total number of comparison is (n-1)+(n-2)+...+2+1 = n(n-1)/2, which for large n is $O(n^2)$. For quicksort, Assume the elements are randomly ordered so that the choice of v to divide the list is random. Quicksort divides the list: at the first level there are two sublists approximately half the size of the original; at the second left there are four sublists approximately 1/4 the size, and so on until the sublists are of size 1. At the kth level, where $n=2^k$, the sublists are of size 1. At each level all the elements are scanned once. Thus the total effort is proportional to $nk = O(n \log_2 n)$. The minimal possible sorting algorithm takes time $O(n \log_2 n)$ because, for each element, we need k bits of information to identify the element's proper position in the output list.

machine running at 10^6 operations per second, but only about one second on a 1990s Pentium II chip. This notation also makes it easier to see how the running time changes with the problem size. For example, the $O(n^2)$ sorting algorithm would take four times longer to sort a list twice as long.

We can use the categories of complexity expressed in order notation as a way to grade the difficulty of problems. For example, any problem whose algorithms take time O(n) is called "linear"; any problem whose algorithms take time $O(n^2)$ is called "quadratic"; any problem whose algorithms take time $O(n^3)$ is called "cubic"; and any problem whose algorithms take time $O(2^n)$ is called "exponential". The running times of order $O(n^k)$ are grouped under the common heading of "polynomial". All problems for which a polynomial time algorithm exists are grouped into the set P. The table below shows the operation counts and sample real times on a modern computer chip capable of 10^8 operations per second.

Class	**Complexity**	**Number of operations when $n=10^6$**	**Real time at 10^8 ops/sec**
Polynomial			
Linear	O(n)	10^6	0.01 sec
Quadratic	$O(n^2)$	10^{12}	2.8 hours
Cubic	$O(n^3)$	10^{18}	317 years
Exponential	$O(2^n)$	10^{301030}	10^{301015} years

This table illustrates two important points. First, the exponential-time algorithms are "intractable" (worse than "hard") because they require times well beyond the age of the universe no matter how much computing resource we have. Second, the polynomial-time algorithms are "tractable" in the sense that we can solve (or plan to solve) reasonable problems in a reasonable time with enough computing resources. For example, a machine of 10,000 parallel processors might be able to get those 317 years down to about 4 months.

This brings up an interesting question. Would "parallel algorithms" -- algorithms that could create as many copies of themselves as needed to work simultaneously on alternatives -- make hard problems tractable? To avoid the clutter of mechanisms to manage parallel tasks, we use an abstraction called a *nondeterministic algorithm*. Where the parallel algorithm would create multiple copies of itself to simultaneously explore multiple alternatives, the nondeterministic algorithm simply "guesses" the correct alternative. If the right series of correct guesses solves the problem within polynomial time, we say that the problem is "nondeterministic polynomial", or NP.

Note that if we try to implement an NP problem on a parallel computer, the implementation may spawn exponential numbers of processors as it encounters nondeterministic alternatives. The overhead of creating all those processors and

of keeping track of them may be exponential. Just because we know a fast, nondeterministic algorithm does not help us find a fast deterministic algorithm.

This is one of the great open questions in computer science -- whether P and NP are the same -- i.e., whether there is some yet-undiscovered way to find solutions to the NP problems in polynomial time. Typically, the best known algorithms for NP are exponential, but no one has been able to prove that we can or cannot do better.

As part of the investigation of P and NP, algorithms experts have discovered a subset of NP they call NP-complete (NPC). The NPC problems are decision problems -- they yield a yes/no answer -- with the interesting characteristic that any one can be converted to any other with a polynomial-time computation. Thus if any problem in NPC has a polynomial-time algorithm, we would be able to find a polynomial-time algorithm for every other problem in NPC. Since the NPC problems are the hardest problems in NP, such an outcome would show that P and NP are the same.

But this appears quite unlikely. The class NPC contains all sorts of problems arising in everyday life -- including scheduling, transportation routing, circuit layout problems, cryptographic key management, and many optimization problems. Over 3000 problems are known to be NP-complete. No fast algorithm has been found for any of them. This is taken as strong empirical evidence that there are no fast algorithms for any NP-complete problem. (Garey & Johnson).

The knapsack problem is an example of an intractable problem. We are given a set of objects, each with a size s_i and value v_i, and a knapsack of capacity C. The problem is to find out whether there is a subset of the objects that fit into the knapsack and maximize the value of the knapsack. In other words, s_i summed over this subset does not exceed C and v_i summed over this subset is larger than for any other subset. This problem is a model for many real scheduling situations. For example, C can represent a deadline, s_i are the execution times of possible tasks, and v_i are the incomes generated by running the tasks; the problem is to schedule as many tasks as possible to complete by time C and achieve the maximum return. The only known general algorithm for the knapsack problem is exponential-time: generate all 2^n subsets of the n objects, compute the size and value of each, and select the one that achieves maximum value among all those that fit in C. We can make a decision-problem version by saying that we want to know whether there is a way to fill the knapsack with total value larger than V. Our best algorithm still takes exponential time to find such a subset; once a subset is found, it is very easy to prove that it fills the knapsack and has value exceeding V.

Let us summarize the large amount of ground covered above. A problem is computable if an algorithm exists to solve it. The computable problems come in various grades of difficulty. We measure the running time of an algorithm with an order-of-magnitude function of the size of the input. All the problems with algorithms of running times $O(n^k)$ for some fixed k are called a polynomial-time problems and are grouped into the set P. All the problems with running times $O(2^n)$ or worse, but whose solution can be verified in polynomial time, are grouped into the set NP. In the NP-complete subset of NP, a polynomial-time algorithm for any one can be transformed into a polynomial-time algorithm for any other. But because over 3000 practical problems are known to be NP-complete and no one has been able find a polynomial-time algorithm for any of them, we strongly believe that NP and P are different. We will have to settle for algorithms that work on restricted versions of the problem or for algorithms that give approximate solutions.

Turning the Bad into Good

Cryptography is one area where all the bad news about intractability has been put to good use and has produced enormous social value (cf. Harel). Public key cryptosystems are widely used to provide secrecy, authentication, and nonrepudiation. Secrecy means that a message can be transmitted through the open medium (e.g., Internet) but cannot be deciphered by any eavesdropper. Authentication means that we can establish unequivocally who the other party in a communication is. Nonrepudiation means that the author of a document cannot later successfully disclaim authorship. Public-key systems have enabled secure transactions and signed documents, both essential in electronic commerce.

Only one implementation of public-key cryptography has been survived all attempts to break it. The RSA cryptosystem -- named for its inventors Ronald Rivest, Adi Shamir, and Len Adelman -- has been in commercial use for over two decades. In this system, the public key is specified as the product of two large prime numbers and the secret key is the pair of primes. Anyone can encipher a message under the public key, but only the owner can decipher it. Only, that is, if no one can factor the public key into its two primes. Factoring a composite number into its primes is intractable. The only known algorithms for doing it run in times worse than pure exponentials. These algorithms, running on the fastest known (or imaginable) supercomputers, cannot factor a 500-digit public key within the expected lifetime of the universe.

Algorithm experts believe that no polynomial time algorithm will be found for factoring a composite number. It is possible that an alternative form of computing, quantum computation, can give sufficient speedup to threaten the RSA system, but no one knows how to build a quantum computer and no one has any project when (or if) such a computer will be built. For the time being, at least, the RSA cryptosystem is secure.

Heuristic Algorithms

The fact of the matter is that we recognized early on that some problems are extremely hard to solve if we insist on exact, correct answers. But if we are willing to accept answers that may be approximate, we can often do a lot better. Beginning in the 1950s, we began to use the term "heuristic algorithm" for this purpose. The adjective "heuristic" means approximate, based on operating principles that do not always give exact answers.

Early examples of heuristic algorithms arose in chess-playing programs. An ideal chess machine operates as follows. It enumerates a tree of all possible future chessboards corresponding to all possible sequences of moves from the current board. It finds the paths in the tree that lead to wins for the machine. It takes as its next move the first step in the best path. The problem with this idea is that the enumeration algorithm takes exponential time. Even the most powerful computers and massive storage systems can only look ahead a few moves. There is not enough time to complete the enumeration and select the best move. So a typical chess program uses a compromise. It enumerates boards only a few moves ahead (say 3 or 4). It assigns a score to each board based on rules-of-thumb known to chess experts. It selects as its current move one that leads to the strongest board within a few moves. The heuristic that permits the "pruning of the tree" to a manageable size is the rule that scores a board. Chess programs today use such heuristics and regularly beat players of advanced skill levels.

The only sure way to find an optimal solution for many practical problems is to search through the entire set of all possible solutions. In many cases, the space of possible solution is so large that only a minute fraction of it can be explored. Is there any way to organize the search to give a high probability of finding a near-optimal solution? A typical heuristic for this purpose is to take a trial solution and see if it can be improved by making simple changes. When the trail solution cannot be improved by further changes, the algorithm stops and proposes it as the answer. (The box illustrates with the knapsack problem.)

Simple heuristics can do well with the knapsack problem. The objects are ordered by decreasing value of the ratio v_i/s_i. Then the knapsack is filled starting with the object of highest ratio and continuing until no more objects fit. This heuristic packs the knapsack first with the objects that bring the highest value per unit of object size. A new trial solution can be obtained by picking a pair of objects, one in the knapsack and one outside, and swapping them. If the new set fits and increases the value, we have a better trial solution. The construction of the initial trial solution takes time O(n). The search for a better trial solution takes time $O(n^2)$. This gives us a heuristic algorithm for filling the knapsack within time $O(n^2)$.

Genetic search algorithms mimic a Darwinian principle to achieve the same end. Instead of evolving one trial solution toward the optimum, we evolve a population of candidates that reproduce and mate through several generations until an optimal solution appears. First, the problem is formulated in such a way that any solution can be encoded in a string of binary bits, all of the same length. Each string can be assigned a fitness value based on how well the corresponding problem solution meets the stated goal. Starting with a population of strings, a new population of the same size is generated in two stages, called reproduction and mating. In the reproduction stage, each individual's probability of being reproduced is proportional to the string's fitness. The fittest individuals tend to produce the most copies at this stage.[‡]

The mating stage simulates recombination of genetic elements. Mating between a pair of strings begins by selecting a random integer between 1 and one less than the string length; this defines a crossover point. Two strings are mated by joining the prefix of one with the suffix of the other relative to the crossover point. Because the algorithm is probing an enormous search space with a minute set of candidates, there is a danger of converging on a solution that is only locally optimal but not globally optimal. To avoid such traps, mutations are introduced during the mating stage: each binary bit has some small probability of being reversed during genetic recombination.

Ken DeJong, who has empirically studied genetic algorithms applied to optimization problems, reports that populations of 50 to 100 individuals taken through 10 to 20 generations have a high probability of including optimal or near-optimal individuals. Mutation probabilities on the order of 1/1000 per bit are enough to prevent the search from locking on to a local optimum.

[‡] One way to arrange this is to create a roulette wheel whose circumference is divided into one segment for each string in the population. The length of each segment is proportional to the string's fitness. The wheel is spun many times, each time yielding a string to carry forward into the next generation. This process generates a list of copies of a subset of the previous population.

Example of genetic algorithm solving the knapsack problem. Consider these objects:

Objects:	1	2	3	4	5	6
Values:	2	4	1	6	3	3
Sizes:	3	2	3	3	2	1

We can represent a selection of objects with a string of 6 bits; for example, 011001 means that objects 2, 3, and 6 are in the knapsack. Since $s_2+s_3+s_6=6$, this selection fits. The fitness of a string is 0 if the size is larger than C and is V otherwise; for example, 011001 has fitness $V=v_2+v_3+v_6=8$. A population of N strings, randomly selected to represent N trial solutions, can be transformed to a new generation of N strings as follows. (1) Make two copies of the N/2 most fit strings; discard the rest; scramble the new set of N strings. (2) For each pair of strings, select a crossover integer j; exchange the prefixes of length j for the two strings. We continue to produce new generations until there is no improvement in the highest fitness in the population. The final answer is the most fit individual in the final generation. The chart below illustrates for three generations. In each generation, the first column of strings is either the initial condition or the result of the previous generation; fitnesses are shown in parentheses. The second column is a scrambled list of 2 copies of each of the 3 most fit from the first column. When each pair in the second column is mated at the indicated crossover point, the result is the first column of the next generation. In the third generation, the most fit string is 011101, which is optimal.

011011 (11)	011011 (4)	011001 (8)	011011 (4)	011000 (5)	011101
001011 (7)	011001	011011 (11)	011100	011111 (0)	011001
101100 (9)	101100 (1)	111001 (10)	111001 (4)	111000 (7)	011101
011001 (8)	011001	001100 (7)	011100	011101 (14)	111000
001100 (7)	011011 (2)	011100 (11)	011011 (2)	011001 (8)	011001
111000 (7)	101100	101011 (9)	111001	111011 (0)	111000

Universal Computation and Virtual machines

Perhaps the aspect that most distinguishes computing machines from other machines is their universality. Turing showed how to construct a universal machine -- one that will simulate any other machine and produce the identical result. If U denotes a universal machine, then U(M,X) computes M(X), the result of applying machine M to input X. To do this, we need to represent the program of the subject machine in a standard notation -- that is, in a simple programming language for Turing machines. This means that any computer can "become" any other simply by loading and executing its software.

The universality of Turing machines has been a strong argument in favor of the Church-Turin g thesis. Every time anyone has proposed a new paradigm for computing, someone else has shown how to get a Turing machine to simulate it. Thus the new paradigm has no new computing capabilities. The new paradigm, however, may be far more efficient at some tasks than others. But the same problems will be noncomputable or intractable.

The universal machine U takes an input tape containing #p(M)#X#, where p(M) is a program M, and outputs the tape #M(X)# if M halts. To do this, U maintains a copy of the "instantaneous configuration" of M, which is a string of the form #XaqsbY#, where q is the current state of the control unit, s is the symbol on the current square of the tape, Xa is the string of symbols to the left of the current square, and bY is the string of symbols to the right. The operation of U is to locate in the program p(M) a statement that says what to do if M is in state q observing symbol s. That statement will specify that a new symbol be written where s was and that the head shall move one square right or left. For example, if the statement says: "(q,s): write s', enter state q', and move left," U updates the instantaneous configuration to #X'q's'abY#. U continues this until it encounters a statement in p(M) that halts the computation. It may come as a surprise that Marvin Minsky found a universal machine with just 7 states. Universal computation is not even complex to set up! This simulation is the essence of a stored program computer.

Algorithms have sometimes been called "virtual machines" because they evoke machine-like actions and because a computer can behave like many different machines according as the algorithm it follows. No other machine is like this: mechanical machines have fixed functions.

Processes

We mentioned earlier that there is an important distinction between a program and a process. You cannot fully understand an algorithm until you understand the actions evoked when a machine follows the algorithm's steps. We call the dynamic execution trace of a program its process to distinguish from the static program code itself. Even though the program is finite, uncertainty about whether the program ever exits its loops translates into uncertainty about whether the process will terminate.

Modern programming languages are structured with statements of simple forms that correspond to readily-visualized processes. For example, the form "S1; S2" means that statement S2 follows S1 in the text, and that the process will complete all the steps of S1 before starting S2. The form "if C then S1 else S2" means that the process will evaluate the Boolean condition C and then execute S1 if C is true or S2 if C is false. The form "while C do S" means that the process contains a loop that repeatedly tests C and executes S each time it finds C true. Programs that restrict their statements to these forms are often called "structured programs," a term coined by Edsger Dijkstra around 1970. Structured programs are generally much easier to understand than programs that specify jumps to any statement at any point.

Modern computers treat processes as entities as real as programs. When you tell the computer to execute a command, the operating system creates a process running the program for that command. The operating system keeps track of all the processes you have created and are running on your behalf; and, if there are other users on the system, it does the same for them. Operating systems contain programming statements for manipulating processes. For example, "fork" is used to say "create another process, initially identical to the one doing the creating; "wait" means to pause until a signal is transmitted by another process and "signal" is used to send such a signal.

Because processes do the actual work of an algorithm, we have adopted programming practices that make the process dynamics immediately apparent from the program text and we have endowed operating systems with the capacity to manipulate both processes and programs.

Self-Reproducing Machines and Artificial Life

Many of the earliest designers of computers wondered if computers would one day become intelligent. It is not surprising that this question came up because the logic of computers seemed to imitate human rationality at its best, and because it seemed possible to write computer programs that performed tasks most people at the time considered intelligent. (Checkers is an example.)

The question has always provoked controversy. Many early designers looked to models of neurons for guidance on how to structure computers and organize computations. John von Neumann wrote about self-reproducing automata, which he considered a prelude to life and intelligence. Alan Turing thought that a computer would be able to fool a human into thinking that the computer itself is human for at least five minutes by the year 2000 (that was fifty years after he made his prediction). Around the time Turing was speculating about intelligent

machines, Norbert Weiner wrote *God and Golem*, raising doubts about whether machines were ultimately capable of thoughtful actions.

Although these pioneers thought it would be half a century before we would know for sure whether we could build intelligent machines, they did not have to wait as long to speculate about whether machines can have life. One of the aspects they examined was self-reproduction. Self-reproduction is an aspect of life, but it is not a sufficient condition for biologists to say that life is present.

It is easy to imagine a machine that constructs other machines. Today's Computer Aided Manufacturing (CAM) systems are prime examples. A CAM system is given a specification of an object that can be build from a given inventory of parts. The machine retrieves parts of the inventory, positions them, and assembles the object. Putting this into notation, we can say that CAM(I,d(O)) creates an object O from the inventory I based on the description d(O).

Now suppose that the inventory of parts is sufficiently rich that we could build another CAM from those parts. Under what circumstances could we actually get CAM to build a copy of itself? That is, can we set things up so that CAM(I,d(CAM)) is another CAM? This is not immediately obvious because CAM would have to be sufficiently complex; is a constructor necessarily more complex than the objects it constructs? The answer is that CAMs can construct objects of greater complexity than themselves. A seven-state universal Turing machine can simulate the computation of any other machine, no matter how complex. A CAM can keep gluing parts from the inventory together until it constructs an object containing many more parts than itself.

A universal machine constructor U takes d(M), the description of a machine M, and builds M from parts in an inventory. Thus U(d(M)) = M. Let us augment U with a duplicator D and a controller C. The controller tells D to make a copy of the input d(M) and save it; then it tells U to construct M; then it places the copy of d(M) in the input tray of M and stops. What we now have is a machine U+D+C that starts with input d(M); when it stops it has produced a machine M with input d(M). What happens when we set M = U+D+C?

In this case, we start with d(U+D+C) presented to U+D+C, and we end up with d(U+D+C) presented to (the copy of) U+D+C. The machine U+D+C is self-reproducing.

The argument in the box above only shows the *existence* of a self-reproducing machine, but does not tell us how complex such a machine might be. It turns out not to be very complex at all. An ancient sport practiced by experienced programmers is to write short code segments which, when compiled and executed, print out exact copies of themselves. Because these code segments are self-referential (as in the box above), they are not easy to understand.

The CAM system, the universal constructor, and the self-reproducing program all illustrate that the phenomenon of reproducibility is closely linked to universal computation and to self-reference. In his book, *Metamagical Themas*, Douglas Hofstadter gives numerous examples of self-reproducing *sentences*. His favorite:

```
after alphabetizing, decapitalize FOR AFTER WORDS
STRING FINALLY UNORDERED UPPERCASE FGPBVKKQJZ
NONVOCALIC DECAPITALIZE SUBSTITUTING ALPHABETIZING,
finally for nonvocalic string substituting unordered
uppercase words
```

Try it!

Scientific Methods

The words "method" and "methodology" are often used to suggest well formed procedures for accomplishing complex tasks. Are methods algorithms?

A good example is the scientific method, the process of scientific discovery. The method is: Observe (collect data), look for patterns (recurrences), formulate a model, make prediction. Repeat. With this method, the scientist is said to evolve and validate a model for a natural phenomenon that other scientists can verify independently.

The "method" is little more than a general guideline. It is used in a backward-looking description of what was done, but it is not usable as a prescription for scientific action (cf. Latour). It gives an outline of the story after the discovery has been accepted as a scientific fact. The actual process of discovery can be chaotic . Its every stage can demand considerable creativity from the scientist. What should be observed? What model should be used? How should it be validated? How many repetitions are needed before the model can be accepted as scientific fact? Because the steps of the process fail the test of mechanization, the scientific method is not an algorithm. Scientific discoveries cannot be made systematically by machine. (This does not rule out individual machines making discoveries.)

Another example is the software design method(ology), which says that the process of building a software system follows these steps: requirements, specifications, prototype, test. This is not an algorithm for the same reason.

The danger in formulating human creative processes with descriptions that resemble algorithmic procedures is that the unwary observer can be fooled into thinking that one day, when we develop sufficient understanding and computing power, these procedures will become algorithms. This appears quite unlikely.

Information and Knowledge

The term *information* has come into wide use. We live in the Information Age and we use Information Technology. Our economy depends on information signaled through the marketplace. Our biology depends on information encoded into our DNA. Our learning depends on what information we can acquire. In computing we use information in at least two distinct ways: the bits processed by an algorithm, and the interpretation assigned by a human observer on seeing the output of an algorithm. In the first sense a better word is *data*, as is the early term "data processing." In the second sense information is subjective and depends on the observer. It is a social phenomenon, not a scientific phenomenon.

In the 1940s Claude Shannon developed a theory of information for telecommunications. He offered a precise mathematical definition of information and noted that it followed a formula like entropy in thermodynamics. Unfortunately the entropy definition works only in the narrow domain of telecommunications and does not apply in most other areas of computer science and engineering. Many people find it confusing that a scientific and engineering discipline claims to be based on a principle that cannot be precisely defined.

In social practice and education, we tend to separate the terms data, information, and knowledge as follows:

> *data* are sequences of symbols arranged in patterns according to a set of syntax rules
>
> *information* is an interpretation of a human observer that data answer a question or enable a new action
>
> *knowledge* is the capacity for a human being to act effectively in a given domain

An important point is that knowledge must be embodied before a human being can be judged competent. Embodiment takes time and, often, considerable practice. Models of education and learning that claim to be based on information transfer and application can be grossly misleading and can produce bad curricula and public policy.

References

Brown, John Seely, and Paul Duguid. 2000. *The Social Life of Information.* Harvard University Business Press.

Davis, L. (ed). 1991. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold.

Goldberg, David E. 1989. *Genetic Algorithms.* Addison-Wesley.

Harel, David. 2001. *Computers, Ltd: What Computers Really Can't Do.* Oxford University Press.

Hofstadter, Douglas. 1984. *Gödel, Escher, Bach: The Eternal Golden Braid.* Harper.

Holland, John R. 1975. *Adaptation in Natural and Artificial Systems.* U Michigan Press.

Knuth, Donald. 1968. *The Art of Computer Programming, Vol 1: Fundamental Algorithms.* Addison-Wesley.

Latour, Bruno. 1988. *Science in Action.* Harvard University Press.

Shannon, Claude, and Warren Weaver. 1963. *Mathematical Theory of Communication.* University of Illinois Press.

Talbott, Stephen. 1998. "There is no such thing as information." Netfuture 92. An on-line journal at <http://www.oreilly.com/~stevet/netfuture>.

Turing, A. M. 1936. "On computable numbers, with an application to the Entscheidungsproblem." *Proc. London Math. Soc. 2-42*, 230-265. (Correction *ibid.* 2-43, 544-546.)

