THE TURING OMNIBUS

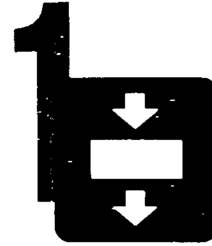
61 Excursions in Computer Science

A. K. DEWDNEY

©1989

COMPUTER SCIENCE PRESS

Rockville, MD



ALGORITHMS

Cooking Up Programs

A program specifies in the exact syntax of some programming language the computation one expects a computer to perform. The syntax is precise and unforgiving. The slightest error in the program as written may cause the computation to be in error or may halt it altogether. The reason for this situation seems paradoxical on the surface: It is relatively easy to design a system that converts rigid syntax to computations; it is much harder to design a system that tolerates mistakes or accepts a broader range of program descriptions.

An algorithm may specify essentially the same computation as a specific program written in a specific language such as BASIC, Pascal, or C. Yet, the purpose of an algorithm is to communicate a computation not to computers but to humans. This is a more natural state of affairs than most people suppose. Our lives (whether we work with computers or not) are full of algorithms. A recipe, for example, is an algorithm for preparing food (assuming, for the moment, that we think of cooking as a form of computation).

ENCHILADAS

1. Preheat oven to 350°F.
2. In a heavy saucepan
    1. Heat 2 tablespoons olive oil.
    2. Sauté
        ½ cup chopped onion
        1 minced garlic clove
       Until golden
    3. Add
        1 tablespoon chili powder
        1 cup tomato puree
        ½ cup chicken stock
    4. Season with
        Salt and pepper
        1 teaspoon cumin
3. Spread sauce over tortillas.
4. Fill centers with equal quantities of
    Chopped raw onion
    Chopped mozzarella cheese
5. Roll tortillas.
6. Place in ovenproof dish.
7. Pour more sauce over tops.
8. Sprinkle with chopped mozzarella cheese.
9. Heat thoroughly in oven about 15 minutes.

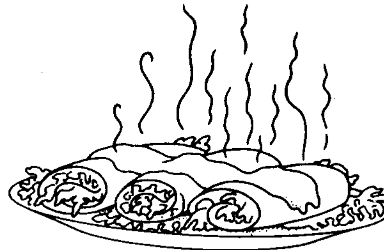


Figure 1 Programmed enchiladas

The purpose of this chapter is not only to acquaint the general reader with the idea of an algorithm, but also to prepare her or him for the manner in which this book presents algorithms. The recipe illustrates some of the conventions to be used as well as some of the broader features shared by most algorithmic styles.

The name of an algorithm is capitalized, as in ENCHILADAS. The individual steps or statements are usually (but not always) numbered. In this respect, one notices a peculiar feature of ENCHILADAS; some statements are indented and appear to repeat earlier numbers. For example, after the first statement labeled 2, there are four statements numbered 1 to 4, all sharing the same indentation. The purpose of the indentation is to make it easier for the eye to recognize that the four steps all take place in the heavy saucepan specified in step 2. If we wish to refer to one of these four steps, we use the step number preceded by the number of the last nonindented step before it. Thus the cook must add chopped onions and garlic in step 2.2. As soon as the saucepan operations have been completed at the end of step 2.4, the statement labels revert to single numbers.

This statement-numbering scheme is used by only a few authors, but it is convenient. Other schemes are possible. The main purpose in numbering statements is to make references to them possible. It is easier to draw a reader's attention to statement 2.3 than to talk about "the step where one adds the chili powder, tomato puree, and chicken stock."

The indentations used in the recipe follow a broader rationale than the indentations used in most algorithms—or programs, for that matter. But statement 2.2 is typical: There is a *continuing* operation (sautéing) that is repeated *until* a certain condition is met, namely, when the onions and garlic have turned golden. Such an operation is called a *loop*. All statements inside the loop are indented.

A major feature shared by ENCHILADAS with all algorithms is the looseness of its description. Glancing through the algorithm, one may notice several operations that are not spelled out. How "heavy" must the saucepan be? When is a color "golden"? How much salt and pepper should be added in step 2.4? These are things that an inexperienced cook (like the author) might worry about. Such things do not bother experienced cooks. Their judgment and common sense fill the obvious gaps in the recipe. An experienced cook, for example, knows enough to remove the enchiladas from the oven at the end of step 9, even though the algorithm does not explicitly include that direction.

Whatever the parallels between cooking and computation, algorithms come into their own in the latter environment. This does not mean that algorithms are incapable of producing the computational equivalent of a good enchilada. Consider, for example, the following graphic algorithm. It produces a mind-boggling variety of pleasing designs on a computer screen:

WALLPAPER

```
1. input corna, cornb
2. input side
3. for i ← 1 to 100
   1. for j ← 1 to 100
          x ← corna + i × side/100
          y ← cornb + j × side/100
          c ← int (x² + y²)
          if c even
             then plot (i, j)
```

This algorithm, when it is translated to a program for a computer with some kind of graphic output device (such as a simple display screen), will enable

users to explore a world of endless wallpaper patterns. Each square section of the plane has its own pattern. If the user types in the coordinates of the lower left-hand corner *(corna, cornb)* of the square of interest, along with the length of its sides, the algorithm draws a picture of wallpaper associated with that square. Strangely enough, if a small square with the same corner coordinates is chosen, a completely different pattern emerges, neither a magnification nor any other transmogrification.

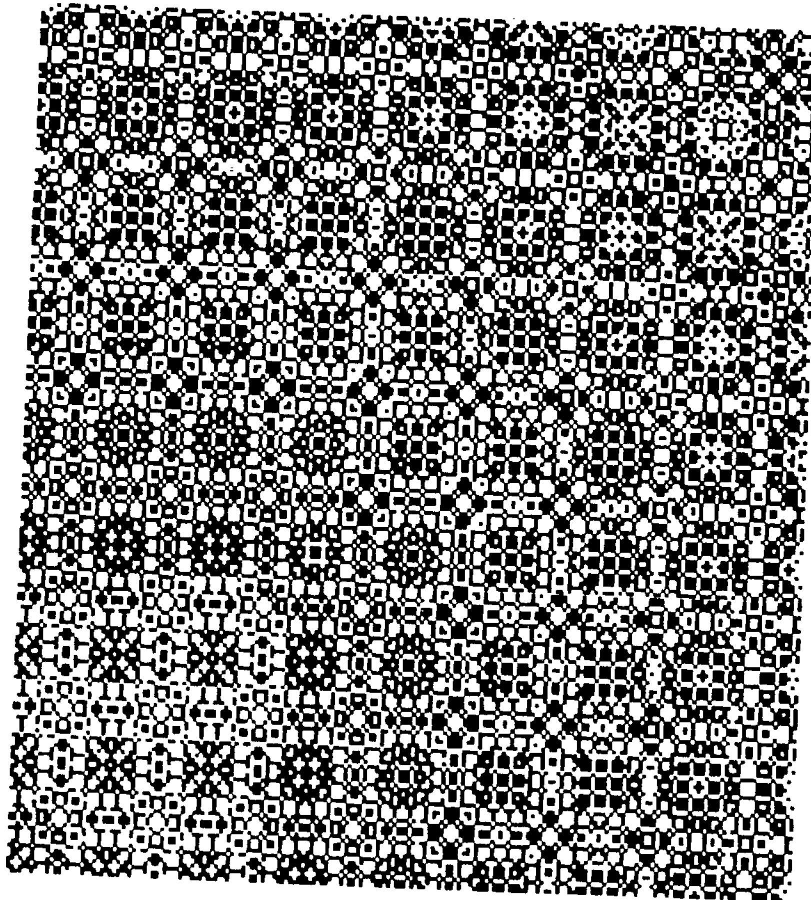


Figure 2 Algorithmic wallpaper

So much for what the algorithm does. How does it work? There are two **for** loops in the algorithm. The outer loop steadily counts from 1 to 100 in step 3, using variable i to hold the current count value. For *each* such value, the inner loop counts from 1 to 100 via j in step 3.1. Moreover, for *each* j value, the four steps that we might label 3.1.1 through 3.1.4 are repeated many times. Each time, of course, there is a new pair of values i and j to work with. First x is computed as a coordinate that is i one-hundredths of the way *across* the square. Then y is similarly computed as a coordinate that is j one-hundredths of the way *up* the square. The resulting point (x, y) is therefore somewhere within the square. As j goes from value to value, one might think of the point moving steadily up the picture. When it reaches the top, $j = 100$ and the inner loop has been fully executed. At this point, i moves to its next value, and the whole inner loop begins all over again at $j = 1$.

The heart of the algorithm consists of computing $x^2 + y^2$, a circular function, and then taking the integer part (int) of the result. If the integer is even, a point (say, white) is plotted on the screen. If the integer is odd, however, no point is plotted for (x, y)—the screen remains dark there.

The WALLPAPER algorithm illustrates several practices followed in this book.

Statements are indented inside the scope of loops and of conditional statements **(if . . . then)**.

Assignment of a computed value to a variable is indicated by a left-pointing arrow.

The computation indicated on the right-hand side of an assignment statement can be any meaningful arithmetic formula.

More than this, we make the statements of an algorithm very general. For example, WALLPAPER might be rewritten as follows:

1. **input** parameters of square
2. **for each** point in the square
   1. compute the c function
   2. **if** c is even, plot it

Such compressed algorithms might serve several purposes. They may represent a stage in the design of a WALLPAPER program in which the programmer begins with a very general, coarse-grained description of the computation to be carried out. He or she then rewrites the algorithm a number of times, refining it at each stage. At some point when WALLPAPER has reached a reasonable level of detail, it can be translated more or less directly to a program in any of a number of actual programming languages, from BASIC to C.

A second reason for writing compressed algorithms lies in the level of communication with another human being. For example, the writer of an algorithm may share a certain understanding with another person about the intention behind each statement. To say "**for each** point in the square" implies a double loop. The "*c* function" might well be understood to mean steps 3.1.1 to 3.1.4 in the WALLPAPER algorithm.

The kind of algorithmic language used in this book and elsewhere freely borrows constructs that are available in a great many programming languages. The following loop types are useful:

for . . . **to**
repeat . . . **until**
while . . .
for each . . .

There are **input** and **output** statements, conditionals such as **if** . . . **then** . . . or **if** . . . **then** . . . **else**. . . . Indeed, readers should feel free to extend the algorithmic language used here to include any language construct that seems reasonable. Variables can be real numbers, logic-valued (boolean) numbers, integers, or virtually any single-valued mathematical entity. One may have arrays of any type or dimension, lists, queues, stacks, and so on. Readers unfamiliar with some of these notions may discover their meanings in the context of whatever chapters use them.

The translation of an algorithm to a program is generally straightforward, at least if the algorithm is reasonably detailed. For example, the WALLPAPER algorithm translates readily to the following Pascal program:

```
program WALLPAPER (input,output);
var corna, cornb, side, x, y, c: real;
var i, j: integer;
begin
   read(corna,cornb);
   read(side);
   graph mode;
   for i: = 1 to 100
   begin
      for j: = 1 to 100
      begin
      x: = corna + i*side/100;
      y: = cornb + j*side/100;
      c: = trunc(x*x + y*y);
```



```
         if cmod2 = 0 then plot (i,j,1)
         end
      end
      textmode;
end
```

Insofar as an algorithm represents computation, one might say that *The Turing Omnibus* is almost a book about algorithms. What problems can (or cannot) be solved by algorithms? When is an algorithm correct? How much time does it take? How much memory does it use? Many algorithms, moreover, embody subtle and interesting intellectual questions. The WALLPAPER algorithm is no exception. Why is it, for example, that in so many cases (as in the previous figure) the algorithm produces repeating or nearly repeating patterns?

Every chapter in this book is followed by a small set of problems. Although a great many chapters describe certain algorithms, very few problems suggest that the reader implement the algorithms as actual programs. But the suggestion is always implicit, and students of computing are urged to do precisely this—starting with this chapter.

Problems

1. Write your favorite recipe as an algorithm. How much more complicated does the algorithm become if it is to be used by a novice cook?

2. Rewrite the WALLPAPER algorithm to use not two colors (black and white) but three.

(*Hint:* Even and odd are merely the residue classes of a number taken to modulo 2.)

References

Irma S. Rombauer and Mario Rombauer Becker. *Joy of Cooking*. New American Library, New York, 1964.

David Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley, Reading, Mass., 1987.

28 TURING MACHINES

The Simplest Computers

Turing machines are the simplest and most widely used theoretical models of computing. Far too slow and unwieldy ever to be embodied in a real device, these conceptual machines nevertheless seem to capture everything we mean by the term ***computation***. Not only do Turing machines occupy the top level of the Chomsky hierarchy (see Chapter 7), but also they seem capable of computing any function which is computable by any other conceptual scheme (see Chapter 60). Turing machines, moreover, are simpler than such schemes—from general recursive functions to random access machines (see Chapters 15 and 45).

As with any computational device or model, we must be careful to distinguish "hardware" from "software." In the case of Turing machines, we might visualize the kind of physical setup shown in Figure 90.

An infinite tape composed of discrete cells is examined, one cell at a time, by a read/write head which communicates with a control mechanism inside a box. Under the control of this mechanism, the Turing machine is able to read the symbol in the current tape cell or to replace it by another. The Turing machine is also able to move the tape, one cell at a time, either to the left or to the right.

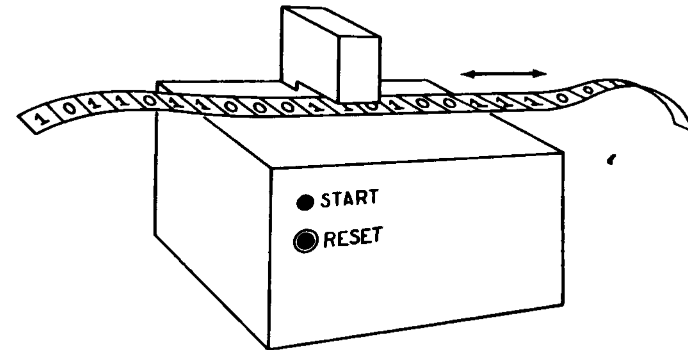


Figure 90 A Turing machine conceptualization

Like the other machines in the Chomsky hierarchy, Turing machines may be treated as language acceptors. The set of words which cause a Turing machine ultimately to halt is considered to be its ***language***. However, as indicated above, Turing machines are much more than this. We consider the function defined by a Turing machine M as follows:

Let x be a string over M's tape alphabet Σ. Place the read/write head over the leftmost symbol of x, start M in its initial state, and allow it to compute without interference. If M eventually halts, then the string y which is found on the tape at that time will be considered as M's output corresponding to the input x. Since M does not necessarily halt for every input string x, M computes a partial function

$$f: \Sigma^* \rightarrow \Sigma^*$$

where Σ^* denotes the set of all strings over Σ. Under this scheme, all sorts of possibilities for Σ, the tape alphabet, may be envisaged. For example, Σ may be used as a basis for the representation of integers, rational numbers, alphabetic words, or virtually any other class of objects suitable for computation.

Strictly speaking, a Turing machine is essentially the same thing as its ***program***. Formally, such a program M is a set of quintuples of the form

$$(q, s, q', s', d)$$

where q is the current state, s is the symbol currently under the read/write head, q' is the state which M is next to enter, s' is the symbol to be written in place of s, and d is the direction in which the read/write head is to move relative to the

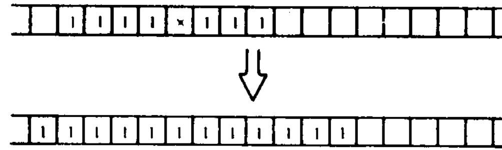


Figure 91 Multiplying two unary numbers

tape. The states come from a finite set Q, the symbols from an alphabet Σ, and the read/write motions from $D = \{\text{left, right, stop}\}$.

A Turing machine program, given a particular input string, will always start in a specially designated *initial state* $q_0 \in Q$ and may be caused to halt in a number of ways. The method we use to halt a program involves the use of *halting states*: When M enters a halting state q', all computational activity ceases. Naturally, q' never appears as the initial symbol in any of the quintuples of M's program.

A Turing machine program P shown below is for multiplying two numbers in unary form. Under this representation, a number m is represented by m consecutive 1s on the tape. Initially, P's tape will contain the two numbers to be multiplied separated by an X symbol. Finally, P's tape will contain the product of m and n in unary form. The initial and final tapes are shown in Figure 91.

As in other conceptual machines, there are several ways of representing a Turing machine program, the most direct being simply a list of P's quintuples:

MULTIPLICATION PROGRAM

$q_0,1,q_1,1,L$	$q_6,1,q_6,1,L$
$q_1,\not{b},q_2,*,R$	$q_7,*,q_7,*,R$
$q_2,\not{b},q_3,\not{b},L$	$q_7,1,q_7,1,R$
$q_2,*,q_2,*,R$	q_7,X,q_5,X,L
$q_2,1,q_2,1,R$	q_7,A,q_7,A,R
q_2,X,q_2,X,R	$q_8,\not{b},q_3,\not{b},L$
q_2,A,q_2,A,R	$q_8,1,q_8,1,R$
$q_3,1,q_4,\not{b},L$	q_8,X,q_8,X,R
q_3,X,q_4,X,L	$q_8,A,q_8,1,R$
$q_4,1,q_4,1,L$	$q_9,*,q_{10},\not{b},S$
q_4,X,q_5,X,L	$q_9,1,q_9,1,L$
$q_5,*,q_8,*,R$	$q_{10},\not{b},\textcircled{q_{11}},\not{b},S$
$q_5,1,q_6,A,L$	$q_{10},1,q_{10},\not{b},R$
q_5,A,q_5,A,L	$q_{10},X,q_{10},\not{b},R$
$q_6,\not{b},q_7,1,R$	$q_{10},A,q_{10},\not{b},R$
$q_6,*,q_6,*,L$	

The most readily understandable representation of a Turing machine program is the state-transition diagram. Such a diagram can often be simplified even further by labeling certain states as "left-moving" or "right-moving" and omitting all transition arrows from a state into itself where the symbol read is rewritten and the movement of the read/write head is inferred from the direction associated with the state. Figure 92 shows the state-transition diagram for P.

The first action of P, on being confronted with two unary numbers to multiply, is to move one cell to the left from the leftmost 1 and to write an asterisk there, entering state 2. Program P remains in state 2, continually moving to the right until the first blank is arrived at. From state 3 P changes the 1 beside this to a blank (state 3 → state 4) and moves left until it encounters the X sign. Now in state 5, it enters a three-state loop in which each 1 in the left-hand unary number is changed to an A and a corresponding 1 is written in the first available cell to the left of the asterisk. In this way, P simply writes a string of m ones. When P finally encounters the asterisk, the string is finished and P moves to state 8, where it changes all the A's back to 1 and then continues moving to the right until a blank is encountered (state 8 → state 3). Here, the cycle begins again, the next 1 in the string of n ones is blanked, and P reenters the three-state loop in order to write another string of m ones adjacent to the string it had just written. In this manner, n adjacent strings of m ones are written, completing the formation of the product $m \times n$ in unary form. In the final stage of its computation, P erases the asterisk (state 9 → state 10) and moves to the right, erasing all symbols (denoted by —) until it reaches the first blank. Then it halts.

Turing machines were first described by Alan Turing whose role as one of the founders of computer science this book celebrates. Turing conceived of his machine in the mid-1930s in an effort to write down in the simplest possible terms what it means for a computation to proceed in discrete steps. Turing suspected that his machine was the most powerful conceivable theoretical

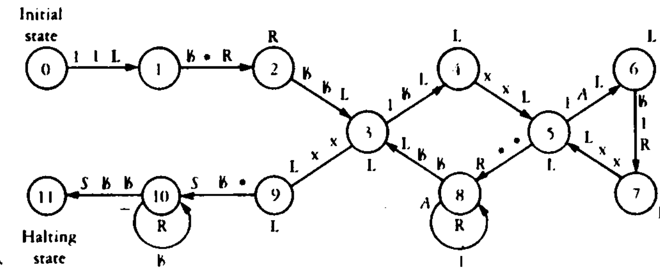


Figure 92 The unary multiplication program

model when he noticed that it was equivalent to a number of other conceptual computing schemes. Another factor which may have given him confidence in this thesis was the robustness of Turing machines. Their alphabets, state sets, and tapes can be changed radically without affecting the power of the class of machines so obtained.

For example, the alphabet of a Turing machine may contain as few as two symbols and still compute anything which machines with larger alphabets can compute. The number of states can also be limited to just two without altering their power! Naturally, there is a trade-off between the number of states and the size of alphabet: Given an arbitrary Turing machine program P employing m states and n symbols, a two-symbol program which mimics P must, in general, have far more than m states. Conversely, the two-state program which mimics P must have far more than n symbols.

One of the most robust features of Turing machines is the variation in the kind and number of tapes they may have without their computational powers being affected. Turing machines may have any number of tapes, or they may get along perfectly well on a single, semi-infinite tape. This is an infinite tape which has been cut in half, so to speak. If the right-hand half is used, then a Turing machine with this sort of tape is not permitted to move beyond the leftmost cell.

A ***multitape Turing machine program*** is not a set of quintuples, but a set of $(2 + 3n)$-tuples, where n is the number of tapes being used. The "instructions" of such a program have the following general form:

$$q, s_1, s_2, \ldots, s_n, q', s_1', s_2', \ldots, s_n', d_1, d_2, \ldots, d_n$$

where q is the current state and $s_1, s_2, \ldots, s_n$ are the symbols currently being read (by n read/write heads) on the n tapes. Under these conditions, this $(2 + 3n)$-tuple tells us that the machine must next enter state q', write the symbols $s_1', s_2', \ldots, s_n'$ on their respective tapes, and move the corresponding read/write heads in directions $d_1, d_2, \ldots, d_n$, where, of course, each d_i may be L, R, or S as in the one-tape machine described earlier.

Although they cannot compute anything which a one-tape machine cannot compute, multitape machines can often compute much more efficiently. For example, given the task of forming the product of two unary numbers, there is a three-tape machine program which does this in approximately $n \times m$ steps. The program presented earlier requires in the order of $n^2 \times m^2$ steps!

We close this chapter with a demonstration that a one-tape Turing machine can do anything that an n-tape machine can do. More precisely, we show that given any n-tape Turing machine program M, there is an $(n-1)$-tape Turing machine program P which does precisely the same thing. The basic construction can then be reapplied to produce an equivalent $(n-2)$-tape machine and so on. Ultimately a one-tape machine that is equivalent to M results.

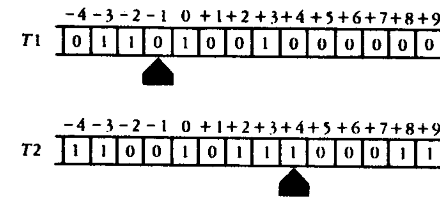


Figure 93 The two tapes of M

For the sake of simplicity, suppose that M is a two-tape Turing machine. Denoting the two tapes by $T1$ and $T2$, select an arbitrary cell in each as cell 0 and number all cells to the right and left of these cells with positive and negative numbers, respectively (Figure 93).

On a new tape, number the cells in pairs 0,0, then +1,+1, and so on, placing the symbol from the ith cell of $T1$ in the first cell of the ith pair and placing the symbol from the ith cell of $T2$ in the second cell of the ith pair (Figure 94). It is convenient to assume that M's tape alphabet consists only of the binary numbers 0 and 1. The positions of the two read/write heads are indicated by alphabetic symbols A and B. These symbols serve only to indicate 0 and 1, respectively. Which head (1 or 2) they represent is kept track of by the Turing machine M' that we will presently construct.

The general idea behind the construction of M' is to have its single read/write head mimic the read/write heads of M by alternating between the two markers. Each marker is therefore moved two cells at a time since cells that are adjacent on one of the two original tapes of M are now two cells apart on the single tape of M'. The new Turing machine M' will be specified in terms of a state-transition diagram that has two symmetric halves. In one half, the one in Figure 95, the read/write marker for tape 1 is assumed to lie to the left of the read/write marker for tape 2. If, in the course of its simulation of M, the two markers of M' should happen to cross, a transition takes control from the first half to the second; here, the tape 1 marker is assumed to lie on the right of the tape 2 marker.

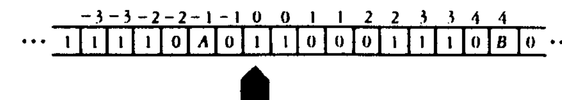


Figure 94 Merging the two tapes into one

The first step in the operation of M' is to decide which 8-tuple of M must be "applied" next:

$$q, s_1, s_2, q', s_1', s_2', d_1, d_2$$

Given that M is supposed to be in state q, we assign the same state to M' and enter a portion of the state-transition diagram that looks like a decision tree. The read/write head of M' is currently scanning the left-hand marker (Figure 95).

In this and subsequent diagrams, some simplifying conventions are introduced: If no symbol appears in the middle of a transition arrow, the symbol at the beginning is merely rewritten. A state labeled L or R is left-moving or right-moving, respectively. In other words, M' keeps moving its read/write head in the indicated direction, remaining in that state, until a symbol appearing on one of its outgoing transition arrows is encountered. Arrows with two symbols at their beginning actually indicated two arrows, one per symbol.

The machine M' first branches to one of the two right-moving states depending on whether the left-hand marker is an A (0) or B (1). It continues moving to the right in either state until it encounters the right-hand marker. Here, a similar decision is made, and the read/write head now moves back to the left-hand marker prior to a transition out of the diagram, so to speak.

At this stage there is enough information to specify which transition

$$q, s_1, s_2, q', s_1', s_2', d_1, d_2$$

is to be simulated by M. The symbols s_1 and s_2 are both known. They are represented by the values (A or B) of the two markers. The diagram that handles the writing of s_1', s_2', as well as the marker movements indicated by d_1 and d_2, has a very simple structure (Figure 96).

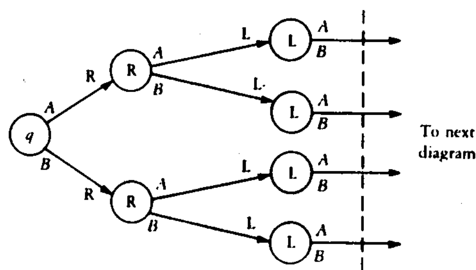


Figure 95 Reading the two markers

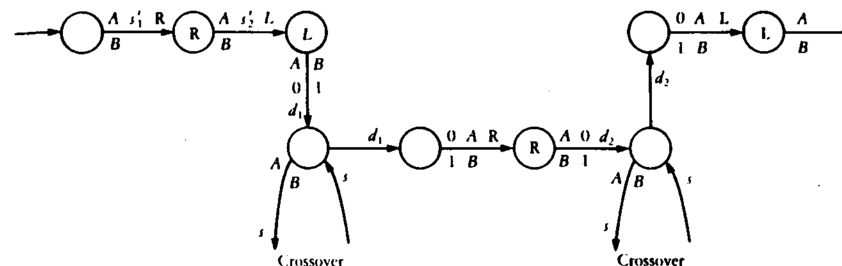


Figure 96 Writing and moving the markers

At the left-hand marker, M' replaces A or B by an alphabetic version of the symbol s_1', then moves to the right until the right-hand marker is encountered. This marker is replaced by an alphabetic version of s_2', and the read/write head of M' moves back to the first marker, replacing it by the corresponding numeric symbol. Then M' moves one step in direction d_1 and checks to see if the current cell holds an A or a B. If so, the markers are about to cross: An appropriate transition thus carries M' into the other half of its diagram, the one for which marker 1 lies to the right of marker 2. Naturally, there is a corresponding transition to this state from the corresponding state of the other half diagram. In any event, M' makes another move in direction d_1 and replaces the numeric symbol in this cell by its alphabetic counterpart. The left-hand marker has been moved two cells in direction d_1.

Next M' carries out the corresponding operation on the right-hand marker and finally moves back to the first marker. As soon as this marker is encountered, M' undergoes a transition that takes it out of the diagram above. Which state does M' enter next? It enters q', a state "borrowed" from M in precisely the manner that q was borrowed. Now M' is ready to pass through another decision tree of states exactly like the one in the first diagram.

Besides showing that n-tape machines are no more powerful than single-tape machines, this construction is very useful in simplifying much abstract theory of computation by allowing us to confine the discussion of what computers can or cannot do to the subject of one-tape Turing machines. For example, in Chapter 60, we use a three-tape machine to simulate a seemingly more powerful abstract device, the random-access machine. By converting this three-tape Turing machine to a one-tape machine by the methods of this chapter, we would obtain a one-tape Turing machine that simulates the random-access machine in question.

Problems

1. Show that a Turing machine with a semi-infinite tape can carry out any computation of which the standard Turing machine is capable.

2. Supply the details for the conversion of an n-tape Turing machine to an equivalent $(n-1)$-tape machine by adapting the 2-to-1 conversion to move complex tuples of the n-tape machine.

3. Write a three-tape Turing machine program for multiplying two binary numbers.

References

M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, Englewood Cliffs, N.J., 1967.

H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, Englewood Cliffs, N.J., 1981.

THE FAST FOURIER TRANSFORM

Redistributing Images

When one is confronted with a computational problem involving a certain mathematical object X, it is sometimes possible to transform X to another object and to solve a (related) computational problem on the newly transformed object. For example, when X is a real number, one may take its logarithm before finding its product with another number Y; instead of multiplying X and Y, one adds their logarithms, taking the antilogarithm of the result:

$$X \cdot Y \rightarrow \log X + \log Y \rightarrow \text{antilog}\,(\log X + \log Y) = X \cdot Y$$

When X is a real-valued function, it is useful in certain problems to take the Fourier transform of X before proceeding. Fourier transforms are used frequently in the processing of images by computer; in Figure 97, for example, the

48

UNIVERSAL TURING MACHINES

Computers as Programs

In Chapter 28, Turing machines were described as a kind of computer, albeit in abstract form. Such terminology can be misleading, however, since Turing machines are normally defined for solving specific problems and are not programmable in the usual sense of the word. Of course, if one thinks of a Turing machine only in terms of its "hardware," that is, its tape, read/write head, finite control unit, and other bits and pieces, then it is reasonable to distinguish between a Turing machine and the programs which run on it. Such programs could be inputted to the Turing machine as a list of quintuples.

In one sense, a programmable Turing machine already exists, at least in the abstract. It is called a ***universal Turing machine,*** and it was first defined by Alan Turing in the 1930s as an abstract model of the most general type of computing device which he could imagine.

A universal Turing machine (Figure 157) has a fixed program permanently

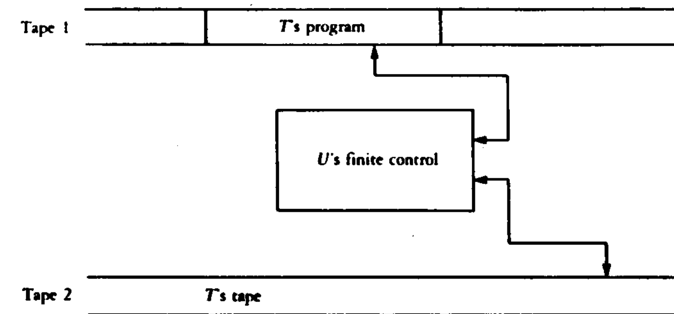


Figure 157 A universal Turing machine

embedded in its finite control unit. This program mimics the action of an arbitrary Turing machine (or program) by reading that program on one tape and simulating its behavior on another.

The universal Turing machine defined here will be called U and will have two tapes. Tape 1 will contain a list of the quintuples defining some Turing machine T, and tape 2 will be blank. On the latter tape, U will mimic the action of T: The initial and final appearances of tape 2 will be identical to its appearance had T been operating on it directly.

As we have already seen in Chapter 28, each two-tape Turing machine can be simulated by some one-tape Turing machine, and this observation applies to universal machines with equal force. In other words, given the construction of U in this chapter, it would be a relatively straightforward matter to construct a one-tape universal Turing machine U^1.

The fixed program inhabiting U's control section is really analogous to interpreter software in a modern digital computer. It is divided into two sections:

1. Given T's current state and input symbol, find the quintuple (q, s, q', s', d) in the description of T that applies.
2. Record the new state q', write the new symbol s' on tape 2, move the read/write head 2 in direction d, read the new symbol on tape 2, and record it beside q'.

The universal machine U expects a particular format for the description of T's program. The quintuples of T and, indeed, all T's tape symbols will use the binary alphabet. Thus, if T has n states, then k-bit binary numbers will be used to index the states of T, where $k = \lceil \log n \rceil$. Given just two tape moves, namely left

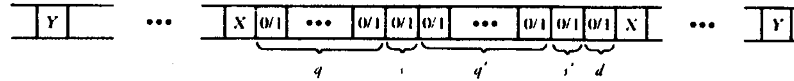


Figure 158 A program to be executed

(=0) and right (=1), each quintuple can be listed as a $(2k+3)$-bit binary number. The quintuples are separated by X's, and two Y's are placed as boundary markers at either end of T's quintuples. In the quintuple shown in Figure 158, the bits marked q and s represent T's current state and input symbol, q' and s' represent T's next state and new symbol, while d represents the direction T's read/write head is to move.

Figure 159 shows a portion of U's interpretive program displayed as a state-transition diagram. This diagram has been made very compact by designating each state as either right-moving (R) or left-moving (L); every transition out of such a state (and, possibly, back into it) involves a movement of U's read/write head in the corresponding direction. Many transitions undergone by U in this and subsequent diagrams are not shown. A "missing" transition simply means that U stays in the current state, moving according to that state's direction and writing whatever it reads. The purpose of the portion of U shown in Figure 159 is to locate the next quintuple to be executed by finding the first two items, the state q and the input symbol s.

Machine U uses the $(k+1)$-bit segment of tape 1 immediately to the left of the left-hand Y marker as a workspace in which to record T's current tape and input symbol. Given that a state label q and input symbol s already occupy this space, the q/s location program is easy to describe. With read/write head 1 on

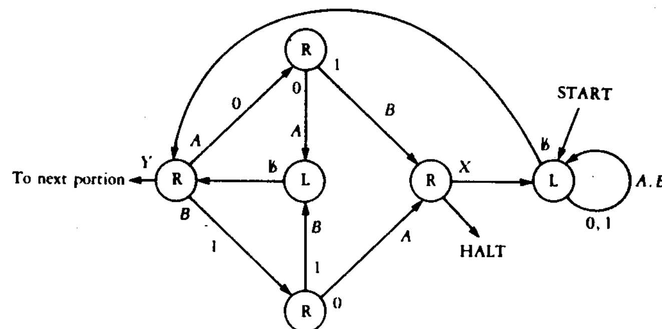


Figure 159 The search for the current quintuple

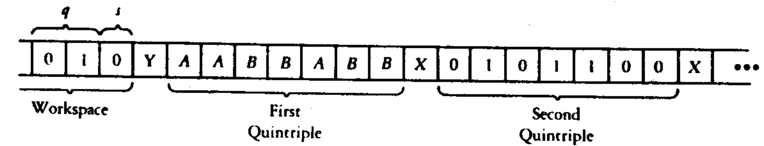


Figure 160 U locates the quintuple.

the left-hand Y marker, U begins in the state labeled START above. It scans to the left, changing each 0 to A and each 1 to B until it encounters a blank. Then it moves to the right, changing the first nonblank symbol it encounters to a 0 or 1 depending on whether that symbol was an A or a B. It then moves to the right, looking for a match for that symbol. The first 0 or 1 bit encountered triggers either a return to the same matching cycle or a return to the starting state, depending on whether a match was found. In all probability, the q/s portion of the first symbol will not match the q and s stored in the workspace. In such a case, U moves right to the first X and then reenters the starting state, where it changes all the 0s and 1s between that X and the left end of the workspace to A's and B's, respectively. Then U attempts to match the q and s in its workspace with the next quintuple. Ultimately, Q either succeeds, taking a transition to the next subdiagram, or halts because it never found a match. In such a case, T would also halt under the usual convention for Turing machines. Figure 160 shows tape 1 after U has found a match for q and s at the second quintuple.

The second portion of U's program records the new state q' in the workspace and moves read/write head 2 according to T's current quintuple. It then records the new symbol in the workspace beside q' on tape 1. In Figure 161, the square

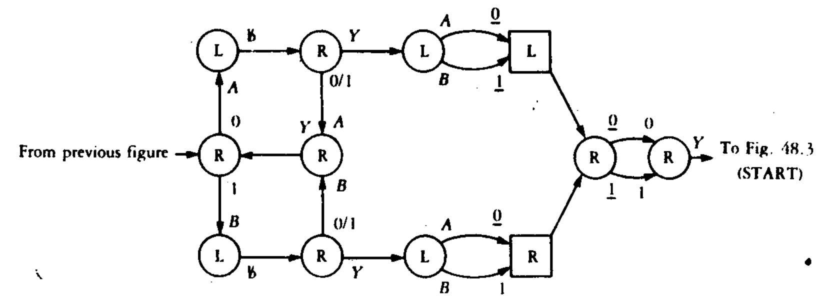


Figure 161 Recording the new state and moving T's read/write head

state symbols indicate moves of read/write head 2. Correspondingly, underlined 0s and 1s indicate symbols read from or written on tape 2.

This portion of *U*'s program is invoked when the first portion has located a quintuple whose q/s component matches the workspace contents. That component has already been rewritten in terms of *A*'s and *B*'s, but the $q'/s'/d$ component still exists as a string of 0s and 1s. This portion of *U*'s program begins with read/write head 2 just to the right of the left-hand *Y* marker.

Scanning to the right, the first binary symbols which *U* finds belong to the next state q'. These symbols are copied one at a time, in terms of *A*'s and *B*'s, into the workspace. When this section of *U*'s program has finished copying q', it next encounters the single binary symbol for s' in *T*'s quintuple. This is the symbol to be written by *T* on its tape. Machine *U* copies s' as an *A* or a *B* into the last (rightmost) cell of the workspace and scans rightward to the quintuple to pick up the last remaining symbol, *d*.

Shown in Figure 162 is tape 1 at this stage in *U*'s operation. When *U* returns to the workspace, however, it finds it has run out of room: it encounters the *Y* marker instead. At this point, *U* "remembers" the value of *d* by being in the upper branch or the lower branch of its state diagram. In either case, *U* scans to the left in order to pick up the s' symbol as an *A* or a *B*, converts this to the corresponding 0 or 1, and then writes it on tape 2, just as *T* would do. Finally, *U* moves read/write head 2 to the left (upper branch of the diagram) or to the right (lower branch), just as *T* would do, and then reads the next symbol from tape 2. Replacing s' in the workspace, *U* shifts right to the *Y* marker and moves back into the portion of program depicted in the first figure.

To set *U* in motion at the beginning of its execution of *T*'s program, it is necessary to place read/write head 1 over the left-hand *Y*. Read/write head 2 is placed over the initial cell of *T*'s tape. The workspace must also be initialized by placing the *k*-bit string for state q_0 (*T*'s initial state) along with the symbol initially scanned on tape 2. And *U*'s program does the rest!

Turing's thesis strikes a close parallel with Church's thesis (see Chapter 60) in declaring that anything which one could reasonably mean by an "effective procedure" is captured by a specific computational scheme, in ths case, Turing machines. A universal Turing machine embodies Turing's thesis in one stroke, so to speak, by simultaneously representing all possible Turing machines and

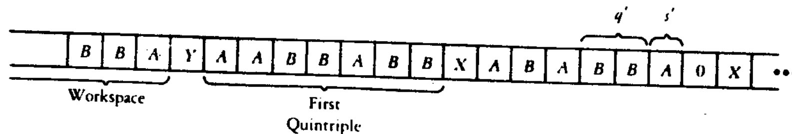


Figure 162 Which way will *T*'s read/write head move?

(by Turing's thesis) all possible effective procedures. It does this on an abstract level in much the same way as a general-purpose digital computer represents all possible programs for it. Its very existence challenges us to explore the range of possible programs; what can it do and what can it not do?

In the theory of computation, the notion of a universal Turing machine serves as a focus for some questions concerning the existence of effective procedures. For example, the halting problem (see Chapter 55) asks for an effective procedure which decides, for each possible Turing machine-tape pair (T, t), whether *T* ultimately halts on *t*. Rather than having to construct such a procedure for all pairs (T, t), it is only necessary to construct one for (U, t), since *t* may initially contain *T* and therefore *U* will halt if and only if *T* does.

Problems

1. As far as *T*'s computation is concerned, *U* is incredibly slow: For each state transition undergone by *T*, *U* undergoes a great many. How many in the worst case? Assume that *T* has *n* states and *m* quintuples in its program.

2. In Marvin Minsky's classic text (referred to below), there is a construction of a one-tape universal machine. Naturally, Minsky's machine is rather more complicated than the one appearing here since a great deal of bookkeeping is required with a one-tape restriction. How much simpler can one make *U*'s state-transition diagram by allowing *U* to have three tapes?

References

Harry R. Lewis and Christos H. Papadimitriou. ***Elements of the Theory of Computation.*** Prentice-Hall, Englewood Cliffs, N.J., 1981.

M. Minsky. ***Computation: Finite and Infinite Machines.*** Prentice-Hall, Englewood Cliffs, N.J., 1967.