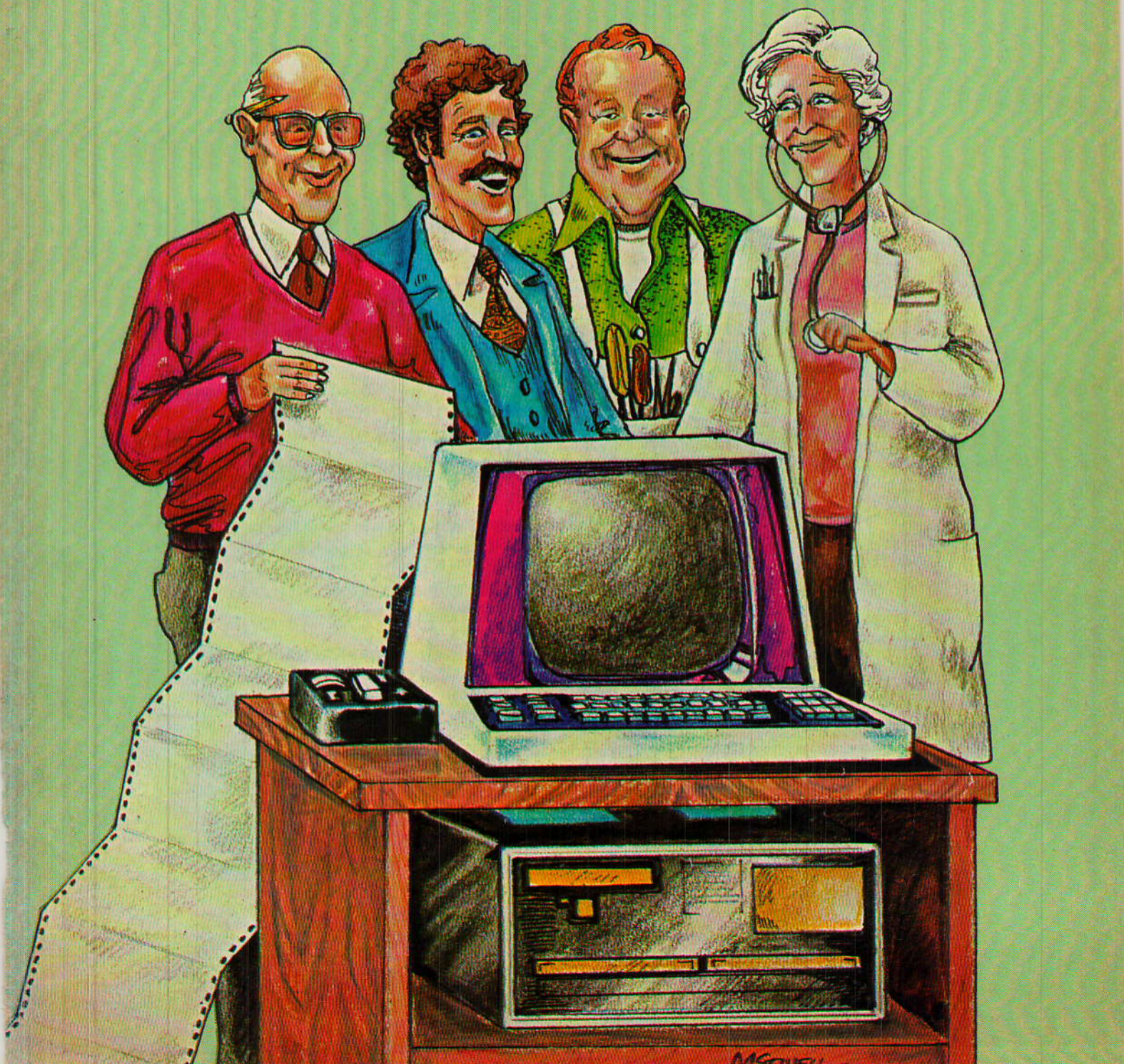
Income Taxes • Simple Game Playing Field • Apple II ...gram

SMALL BUSINESS COMPUTING

$2.00

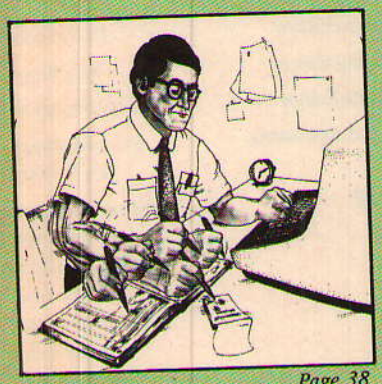
Personal Computing

APRIL 1979

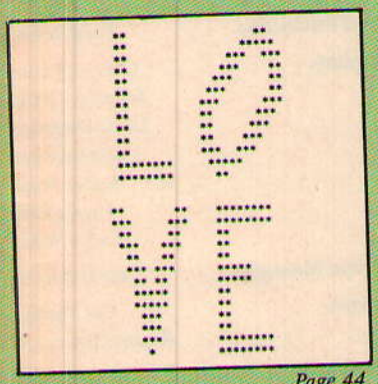


Personal Computing

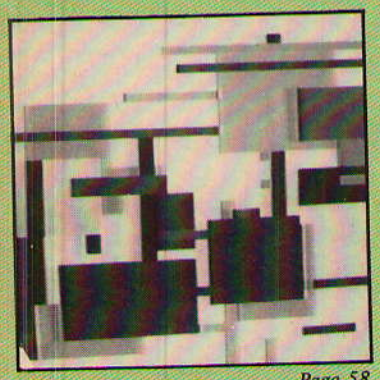
Publication Number USPS 370-770 APRIL 1979 VOL. III NO. 4



Page 38



Page 44



Page 58

Cover Illustration by Mark Sewell

DEPARTMENTS

FEEDBACK 3
RANDOM ACCESS 9
COMPUTER BRIDGE ..53
FIRMWARE FACTS ...56
COMPUTER CHESS ...62
BOOKSHELF 74
WHAT'S COMING UP .76
AD INDEX 111

COVER STORY

Tax Base 22
This comprehensive income tax data base program not only stores tax information for immediate retrieval, but also prints daily and year-end reports helpful when preparing your tax forms. *by Paul Holliday*

Programming Your Computer for a Tax Deduction 34
A tax consultant shows how converting your computer hobby into a small business can shrink your tax bill. *by Mark Battersby*

The Incredible Time Machines 38
The author, a management consultant, shows how small businesses can use computers most effectively — not in the accountant's office, but on the executive's desk. By taking over routine or time-consuming tasks, a computer can free the executive for more important matters such as planning and decision-making. *by Kirtland H. Olson*

How to Add Graphs to Your Computer Output 44
A valuable visual aid, graphs can provide quick analysis and comprehension of statistical material. Use the TAB function to produce graphs for business reports and presentations, or for school papers and projects. *by R. Tickell*

LAUNCHING PAD

Apple II, Artist Extraordinaire 58
Your Apple II can produce original pop-art images on your color television screen. As you rate each picture, the computer modifies it, creating designs expecially suited to your aesthetic tastes. The computer even names its creations for you! *by Raymond T. Vizzone*

DIGGING IN

Recursive Programming in BASIC 16
You don't need a large computer and a complicated language to program recursive procedures. Just use the techniques described in this article, with your micro and BASIC, and you'll add a valuable tool to your repertoire. *by Herbert L. Dershem*

ON THE LIGHTER SIDE

A Simple Game-Playing Field 50
This subroutine, designed primarily for games, lets you use a 100 x 100 playing field without requiring lots of memory or bookkeeping. *by L.D. Stander*

Recursive Programming in BASIC

BY HERBERT L. DERSHEM

Recursive algoithms are anathema to the personal computing programmer. Their reputation suffers from the belief that recursion requires a large, powerful computer and a fancy language. People consider recursive programming so complicated that you need a degree in computer science to understand it. But no one with that much experience would bother with recursion. It is, so the myth goes, highly inefficient, wasteful of resources and only a toy for academics to play with.

But the myths are false. Recursion can be a valuable tool on a personal computer using BASIC and should be in every programmer's repertoire.

Anything is recursive if it is defined in terms of itself. In programming, a recursive subroutine is one that calls on itself. People commonly confuse recursion with iteration; both can be used to solve the same problem.

But recursion and iteration are clearly distinct. A procedure is iterative if the same process is performed many times. A procedure is recursive if in the middle of its execution it calls upon itself. A recursive procedure begins another execution of itself before the original is finished. Such a procedure has more than one execution in progress at a given time; an iterative procedure never has more than one.

Not every language implementation permits recursion. In BASIC, recursion is only possible if it is permissible to call a subroutine from itself and still retain the ability of the original execution to return to the proper point. The following program can test your BASIC for recursion capability.

```
10 REM TEST FOR THE ABILITY OF THE
   BASIC TO PERFORM RECURSION.
20 N=1
30 I=0
40 GOSUB 100
50 IF I<>2*N-1 THEN 170
60 PRINT "RECURSIVE";N-1;"TIMES."
70 N=N+1
80 GOTO 30
100 I=I+1
110 IF I<N THEN 130
120 RETURN
130 GOSUB 100
140 IF I>2*N-1 THEN 170
150 I=I+1
160 RETURN
170 PRINT "NOT RECURSIVE";N-1;"TIMES."
180 END
```

Every call of subroutine 100 increments I by one. The Nth call will result in I being set to N and a return without a recursive call. This will return to each of the previous levels, incrementing I by one on each return. On the final return, I should equal 2*N-1 if the process was carried out correctly. Thus, if the above program runs for a given value of N, the version of BASIC used will allow N-1 recursive calls. Many versions of BASIC have virtually no limit on the number of such calls possible. If the BASIC is not recursive to the appropriate level for some value of N, it will usually respond with some type of diagnostic message rather than arrive at statement 170.

If your computer's BASIC allows recursive subroutines, you now face the problem of writing them. Here's a general outline of a recursive subroutine in BASIC.

1. If the first call, initialize the stack pointer.
2. If termination condition, compute result; decrement stack pointer; return.
3. Do computation.
4. Save necessary values in stack.
5. Increment stack pointer.
6. Recursively call this subroutine.
7. Restore saved values from the stack.
8. Do any remaining computation.
9. Decrement stack pointer.
10. Return.

When you recursively call a subroutine from itself, the variables in the called execution destroy the variables of the same name in the calling execution. To preserve the original values of these variables, save in a dimensioned variable (called a stack) those variables you need to recall later. Suppose your subroutine has three variables (x,y and z) it wishes to save for recall when it returns from a recursive call. The format of the stack dimensioned variable A for this subroutine would be:

A (1) X from execution 1
A (2) Y from execution 1
A (3) Z from execution 1
A (4) X from execution 2
A (5) Y from execution 2
A (6) Z from execution 2
etc.

To keep track of the position in the stack where the current execution saves its values, a pointer is used. This pointer is incremented by the required number, in the above example 3, every time a recursive call is made, and decremented by the same amount on each return.

The general recursive procedure outlined above shows only one recursive call in the subroutine. In general, steps 3 to 8 may be repeated several times before the return at steps 9 and 10.

Let's take a recursive subroutine and see how these steps are implemented. The factorial is the standard first example of recursion – unfortunately, because it also represents a problem where recursion is not the best way to obtain a solution. However, it is the most familiar and simplest of all examples, so why break tradition?

The common definition of N factorial (N!) is iterative:

0! = 1
N! = N* (N-1) * (N-2)*. . .*2*1 for N=1, 2, . . .

But there is also a recursive definition of N factorial:

0! = 1
N! = N * (N-1)! for N=1, 2, . . .

In this case the factorial is defined in terms of itself, but with one escape clause which occurs at 0!. The recursive form of a subroutine to compute N! is:

```
960 REM THIS SUBROUTINE COMPUTES N
    FACTORIAL RECURSIVELY AND
970 REM STORES THE RESULT IN F.
    THE FIRST CALL IS TO 990.
980 REM THIS PROGRAM IS IN RADIO
    SHACK LEVEL I BASIC.
990 S=1
1000 IF N=0 THEN F=1: S=S-1: RETURN
1010 A(S)=N
1020 S=S+1
1030 N=N-1:GOSUB 1000
1040 N=A(S)
1050 F=N*F
1060 S=S-1
1070 RETURN
```

In this program, statement 990 corresponds to step 1 in the general algorithm given earlier. Statement 1000 corresponds to step 2, where the termination condition is N=0. The stack is incremented and decremented by 1 in this program because only one variable, N, is saved when a recursive call is made. No computations in this program correspond to step 3 of the general algorithm, and statements 1010 to 1070 correspond to steps 4 to 10, respectively.

Of course, the simpler iterative version of the factorial subroutine executes much faster:

```
970 REM ITERATIVE SUBROUTINE
    TO COMPUTER N FACTORIAL
980 REM AND STORE IT IN F.
990 REM THIS PROGRAM IS IN
    RADIO SHACK LEVEL I BASIC.
1000 F=1
1010 IF N<=1 THEN RETURN
1020 FOR I=2 TO N
1030   F=F*I
1040 NEXT I
1050 RETURN
```

Another common example of recursion is the computation of the Fibonacci sequence of numbers. The Nth number in the Fibonacci sequence, F(N), is defined in terms of its two predecessors.

F(0) = 0
F(1) = 1
F(N) = F(N-1) + F(N-2) for N = 2, 3, . . .

Your BASIC version of this algorithm is:

```
970 REM SUBROUTINE 990 CALCULATES
    THE NTH FIBONACCI NUMBER
980 REM RECURSIVELY AND RETURNS
    IT IN F. R.S. LEVEL I BASIC.
990 S=1
1000 IF (N=0)+(N=1)
     THEN F=N: S=S-2: RETURN
1010 A(S)=N
1020 S=S+2
1030 N=N-1: GOSUB 1000
1040 N=A(S)
1050 A(S+1)=F
1060 S=S+2
1070 N=N-2:GOSUB 1000
1080 F=A(S+1)+F
1090 S=S-2
1100 RETURN
```

In this example, there are two stack entries for each call level. The Sth entry in A is the value of N for that call level, and the (S+1) st is the value of F(N-1).

Again, as in the case of the factorial, iteration gives a more efficient solution to this problem.

```
970 REM SUBROUTINE 1000 CALCULATES
    THE NTH FIBONACCI NUMBER
980 REM ITERATIVELY AND RETURNS IT IN F.
990 REM IT IS WRITTEN IN RADIO
    SHACK LEVEL I BASIC.
1000 F=1:P=0
1010 FOR I=1 TO N-1
1020   Q=F
1030   F=F+P
1040   P=Q
1050 NEXT I
1060 RETURN
```

Although your first two examples of recursion could be better done iteratively, recursion is the desired technique for many problems because it greatly simplifies the solution algorithm and its implementation in a BASIC program.

Now consider two such problems. The first is the Tower of Hanoi, a well-known problem which is nicely treated in a recursive manner. This problem consists of three pegs, which we will call pegs 1, 2 and 3, and D disks, all of different radius, which can be stacked on the pegs. Initially the disks are stacked on peg 1 in order of decreasing size with the largest disk on the bottom. The problem is to move the disks from peg 1 to peg 2 with the restriction that disks must be moved one at a time from one peg to another, and that no disk may ever be stacked on top of a smaller disk.

The recursive solution generalizes the problem to move D disks from peg E to Peg F by moving the top D-1 disks on peg E to the third peg, moving the one remaining disk on peg E to peg F, and then moving all of the disks on the third peg to peg F. This reduces the problem of moving D disks to two moves of D-1 disks. The recursive algorithm for moving the top D disks from peg E to peg F is:

1. If D=1, move top disk from E to F; return.
2. Let G be the number of the peg which is not E or F.
3. Recursively call this procedure to move the top D-1 disks from E to G.
4. Move the disk on E to F.
5. Recursively call this procedure to move the top D-1 disks from G to F.
6. Return

In our implementation, store the number of disks on peg I in A(I), for I=1, 2, 3. The values that need to be saved when a recursive call is made are E, F and D. Our BASIC version of this algorithm is then:

```
970 REM TOWER OF HANOI SUBROUTINE
    TO MOVE THE TOP D DISCS
980 REM FROM PEG E TO PEG F.
    WRITTEN IN R.S. LEVEL I BASIC.
990 S=4
1000 IF D=1 A(E)=A(E)-1:A(F)=A(F)+1:
     PRINT "MOVE";E;"TO";F:GOTO 1090
1010 G=6-(E+F)
1020 A(S)=E: A(S+1)=F: A(S+2)=D
1030 S=S+3: D=D-1: F=G
1040 GOSUB 1000
1050 E=A(S): F=A(S+1): D=A(S+2)
1060 A(E)=A(E)-1: A(F)=A(F)+1:
     PRINT "MOVE";E;"TO";F
1070 S=S+3: D=D-1: E=6-(E+F)
1080 GOSUB 1000
1090 S=S-3
1100 RETURN
```

Now consider a final useful application of recursion. The quicksort algorithm, – one of the most efficient and widely used sorting algorithms, is very easily programmed recursively.

Suppose we have values stored in A(L), . . ., A(H) and we wish to place the values in ascending order in the same storage locations. The basic quicksort algorithm chooses some arbitrary value from this list, say X=A(K), then rearranges the values so that all values smaller than X are located before it in the list and all values larger than X are located after it. Then X will be at its correct sorted position in the list, say A(I). The same algorithm is then recursively applied with L and I-1 in place of L and H, and then again applied using I+1 and H. When the algorithm is called with L=H, we simply return.

The only part of the algorithm that needs some additional attention is the process of rearranging the list so that X is in its proper position and all other values lie on the proper side of X. Keeping two pointers, I and J, accomplishes this process. I starts by pointing at the first position in the list, L. J points to H. Then pointer I is moved down the list until a value is encountered which is no smaller than X. This value of A(I) should therefore lie below X in the list. Next pointer J is moved up the list until it encounters a number no larger than X. That number is exchanged with A(I), and both are then in the proper part of the list relative to the eventual position of X. The process is repeated until I and J cross. At that point the rearrangement is completed. Figure 1 shows an example of this process.

```
I->12     12     12     12     12     12     12
   75 I->75      3      3      3      3      3
   60     60 I->60 I->60     44     44     44
   13     13     13     13 I->13     13 J->13
X->46 X->46 X->46 X->46 JX->46 IJX->46 X->46
   44     44     44 J->44     60     60 I->60
   94     94     94     94     94     94     94
   49     49     49     49     49     49     49
   95     95 J->95     95     95     95     95
J-> 3 J-> 3     75     75     75     75     75
```

Figure 1

The program to accomplish this process is:

```
960 REM QUICKSORT-SUBROUTINE TO
    REARRANGE A(L) THRU A(H) SO THAT
970 REM ALL VALUES <=X LIE BEFORE X
    AND ALL VALUES >=X LIE AFTER.
980 REM X IS A(INT((L+U)/2). INITIAL
    CALL TO 990 SORTS A(1)-A(N).
990 L=1: H=N:S=N+1
1000 IF L>=H GOTO 1100
1010 M=INT((L+H)/2): X=A(M): I=L: J=H
1020 IF A(I)<X THEN I=I+1: GOTO 1020
1030 IF A(J)>X THEN J=J-1: GOTO 1030
1040 IF I<=J THEN T=A(I):A(I)=A(J):
     A(J)=T:I=I+1:J=J-1
1050 IF I<=J THEN GOTO 1020
1060 A(S)=I:A(S+1)=H
1070 H=J:S=S+2:GOSUB 1000
1080 I=A(S):H=A(S+1)
1090 L=I:S=S+2:GOSUB 1000
1100 S=S-2: RETURN
```

In this program, statement 1000 tests for termination. Statements 1010 to 1050 partition the list into those values smaller than X and those larger than X. The two recursive calls follow statement 1050. Only two values, I and H, are saved during the recursive call.

An illustration of an execution of the quicksort algorithm is given in Figure 2. The original list of ten numbers is found on the far left. The circled numbers are the values used for X, and the number-pairs in rectangles are the values of L and H used to partition the list. Note that we begin with L=1, H=10, X=46. X is then placed by partitioning in the fifth position. Next the process is called for X=3, L=1, H=4 and X=49, L=6, H=10. The process continues in this way until all values are correctly sorted. □

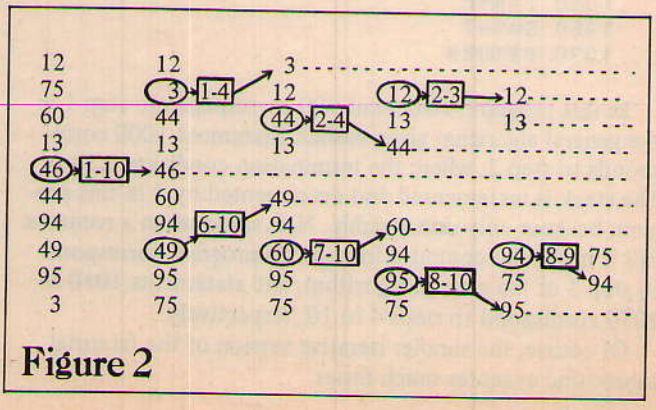


Figure 2