creative computing
reprint series

SORTING, SHUFFLING and FILE STRUCTURES

creative computing
morristown n.j. 07960

A Comparison of Sorts

John P. Grillo
Dept. of Computer Information Systems
West Texas State University
Canyon, Texas 79016

When students in programming courses compare notes, they find that there is a fairly small set of computer problems which are given as programming assignments in practically all courses. Here is a sampling of these Golden Oldies. *The Indian Problem:* If the Indians had deposited the $24 they got for Manhattan Island in 1620 and earned 6% interest compounded yearly, what would that deposit be worth now? *Fibonacci Numbers:* What is the largest Fibonacci number less than a given number? *Grain of Wheat, or Doubled Penny:* Starting with one grain of wheat, or one penny, and doubling the number every day, how many whatevers are there on the 30th (64th) day? *Table Printing:* Produce as output a table of the [squares, square roots, trig functions] of numbers between 1 and 50. *The Sort:* Sort the data provided in ascending (descending) order and print it.

All of these have innumerable variants and all, except for the sort, are based on relatively obvious, simple, or already familiar looping algorithms which show off the computer's ability to handle simple loops. The sort is a different type of problem: (1) the output can be achieved using any of a number of algorithms, but not any are truly easy to understand; (2) the object of the problem is not to produce the output as much as to learn the algorithm and to optimize computer efficiency by minimizing core use and execution time; (3) the algorithm used is often called a production algorithm, one which is used widely in application programming.

All too often students are presented with the simplest algorithm because it is easiest to learn. That is true enough, but unfortunately, that algorithm is the one students tend to use any time they have to sort, just because they know it. This "horseblinders" result would be of no consequence were the algorithm learned the best one, or even one of the better ones. But the algorithm taught, and learned, is usually the worst one, the bubble sort.

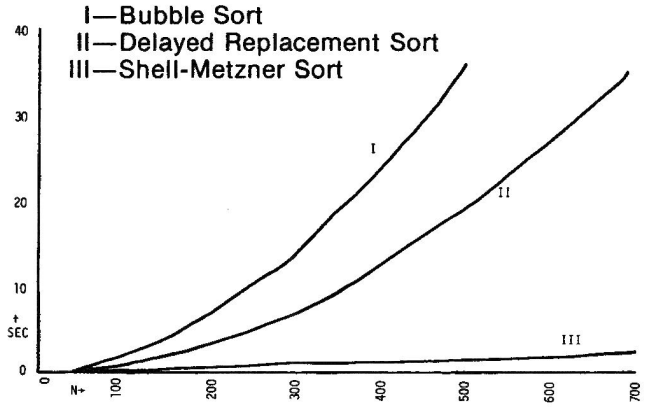
This technique is called bubble sorting because of the way it "floats" the smaller numbers to the top, just like bubbles in a column of water. It might be better called the "trouble sort," though, because of all the machinations that go on at the lower level just to float that number up there.

Slightly better, in terms of efficiency, is the delayed replacement sorting technique. This is really a modification of the bubble sort, except that the smallest of two numbers is not "floated" until it is found to be the smallest of all; whereas the bubble sort floats the smaller of a pair, then checks another pair, the delayed replacement sort checks all pairs and floats only the one found to be the smallest. This greatly reduces the number of executions of the switching statements. The number of pairwise comparisons is exactly equal both in the delayed replacement sort and in the bubble sort, and that number rises exponentially as the number of elements to be sorted rises.

An adaptation by Marlene Metzner (2) of the Shell sort overcomes both difficulties: the number of comparisons is roughly ten times the number of elements to be sorted, and the number of switches is roughly five times the number of elements, if that number of elements is less than 1000. This ratio of comparisons to switches makes intuitive sense, since one would expect that a pair of numbers chosen for possible switching would require switching only half the time.

Appended to this article is a listing of a BASIC program which was used to test sorting algorithms. As an added benefit, the random numbers produced are made to approximate a normal distribution and are truncated. Thus the output from this program can be used as a sample of scores with known statistics. By timing the three methods of sorting using various sample sizes, some estimates of sorting time were calculated. Figure 1 shows graphically the effect of algorithm selection on sorting time.

Figure 1—Observed Sort Times



Figures 2, 3, and 4 are the flowcharts for the three sorting algorithms. All are written to sort a table of N data entries in a table D without use of any additional array space; that is, they are all replacement sorts—array D starts unsorted and becomes sorted. The bubble sort, or Sort I, has as a characteristic feature the use of only two indices, I and J, and no checking of indices except against N, the number of elements being sorted. The delayed replacement sort, Sort II, uses three indices, I, J, and K, and only one of them is compared to N. Note also that in Sort II discovering that D(J) is greater than D(I) does not force a switch; much more index checking is performed first. The Shell-Metzner sort, Sort III, at first glance seems to have regressed to Sort I in that if D(I) is greater than D(L), they are switched. But though this is true, the comparison is performed only after much checking, using not 2, not 3, but 5 indices—I, J, K, L, and M. As a hint in beginning to understand Sort III, consider that the first

Figure 2

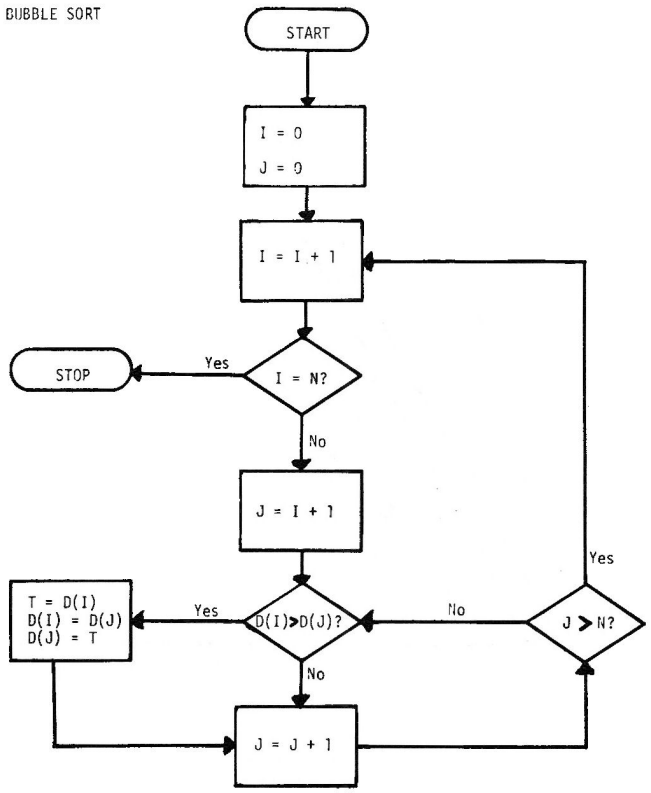


Figure 3

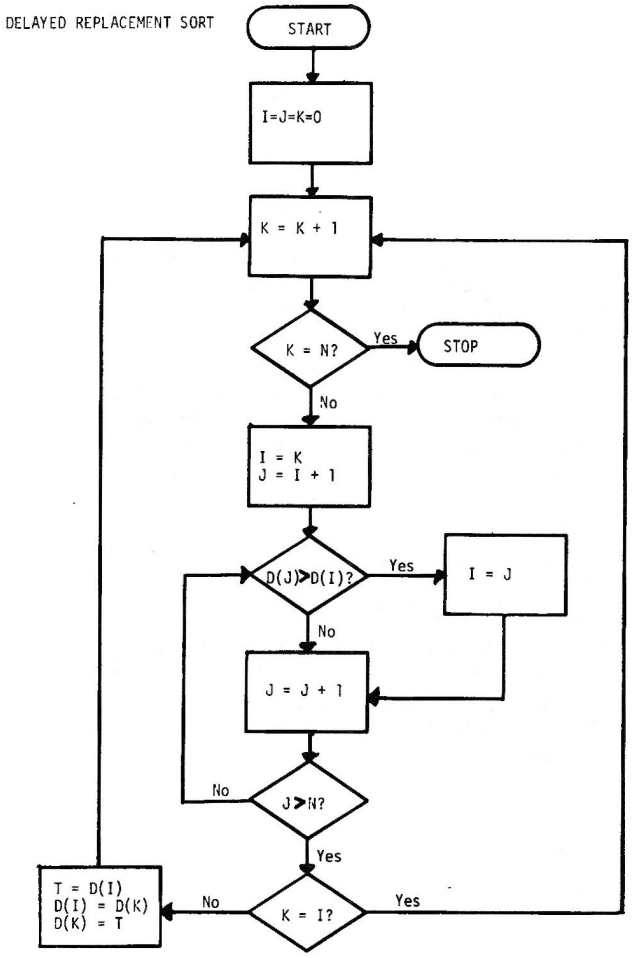
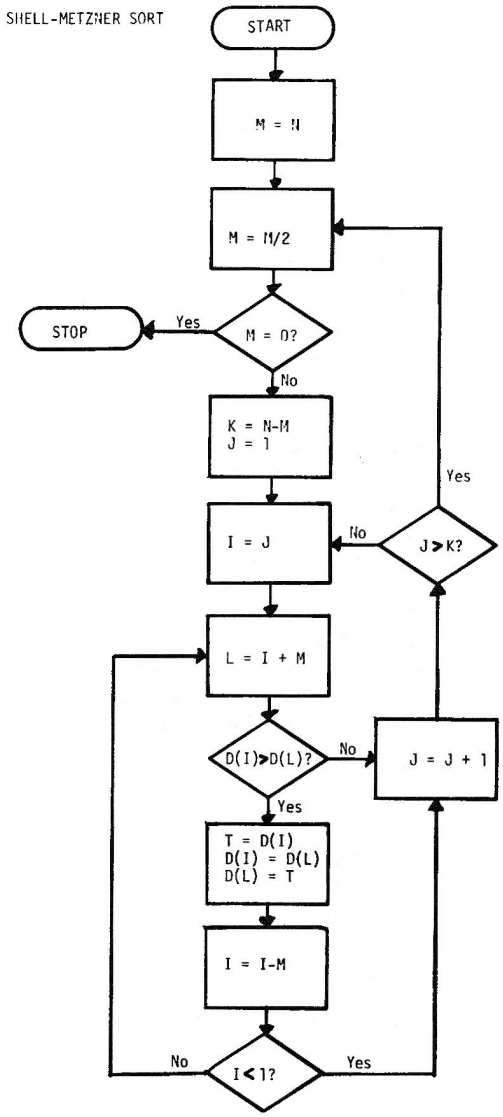


Figure 4



compared pair of 100 numbers is the 1st and 51st; the second is the 2nd and 52nd; etc.

Table I summarizes information on execution of each of the sorts on sets of 10 to 2,000 elements. These elements, or numbers to be sorted, were generated by the program listed in the appendix, so they were normally distributed. The table lists three numbers for each set of elements and each algorithm: T = time of execution in milliseconds on a DECSystem 10 KI processor; S = number of times pairs of elements were switched; C = number of

Table I—Sort Execution Data

		N							Approximate Proportionality to N
		10	20	50	100	200	500	1,000	
SORT I BUBBLE	T	33	84	450	1,700	7,500	34,000	150,000	$.385\ N^{1.84}$
	S	19	100	620	2,700	11,000	63,000	250,000	$.25\ N^2$
	C	45	190	1,225	4,950	19,900	124,750	499,500	$.5\ N^2$
SORT II DELAYED REP'T.	T	17	50	250	830	4,100	20,000	75,000	$.206\ N^{1.84}$
	S	5	17	46	90	190	490	990	N
	C	45	190	1,225	4,950	19,900	124,750	499,500	$.5\ N^2$
SORT III SHELL-METZNER	T	17	34	130	320	600	1,600	3,700	$1.18\ N^{1.18}$
	S	13	34	150	450	930	2,600	5,900	$2\ N^{1.18}$
	C	31	85	320	900	2,100	5,800	13,000	$4\ N^{1.18}$

times pairs of elements were compared. All values in the table were rounded to two significant digits for clarity, except the number of comparisons in Sorts I and II, which are always exact ($(N^2-N)/2$).

One of the effects of sorting normally distributed numbers is evident in Table I: the number of switches in Sorts I and III is less than half the number of compares by an amount equal to the pairs which were equal. That is, almost half of the compared pairs were right to begin with ($A<B$); almost half had to be switched ($A>B$); and some were left alone because they were equal ($A=B$). For this reason the proportionalities shown may increase slightly when these sorts are used on data with very few equal values.

In both Sort I and Sort II all possible pairs of elements were compared once; in 10 numbers the 45 comparisons are: 9 of #1 with the remaining 9; 8 of #2 with the remaining 8; 7 of #3 with the remaining 7; etc., such that the number of comparisons $C = 9+8+7+6+5+4+3+2+1 = 45$.*

In Sort II the number of switches is always less than the number of elements. This is because in this algorithm a switch is executed only when an element has found its place.

In Sort III many elements must be switched more than once, but far fewer compares are executed. One may consider this algorithm to be intelligent enough, so to speak, that it is aware that if $A<B$ and $B<C$ there is no reason to compare A to C; A must be smaller.

Table I also indicates the approximate quantitative relationships between N and C or S for each of the algorithms. A curvilinear regression analysis (1) was performed on the sort times to determine the equations which would predict the sort times given the number of elements. In each of the equations listed below, T is the time in milliseconds, and N is the number of elements. The coefficient and exponent are given to three significant digits only, as this is empirical evidence.

SORT I: $T = .385\ N^{1.84}$
SORT II: $T = .206\ N^{1.84}$
SORT III: $T = 1.18\ N^{1.18}$

Note that the time-saving with Sort II over Sort I is in the coefficient, and that it is in the exponent with Sort III. Figure 5 is a transposed plot of the data in Figure 1, but this time on log-log paper. It is evident that Sorts I and II have equal slopes (thus equal exponents) and that Sort III has a reduced slope.

One cannot resist adding as Table II some sorting times for very large arrays using these three techniques. Of course, one must have available a great deal of memory to perform some of these sorts; and only under special circumstances and with additional merging algorithms can a programmer use these sorting techniques for large disk or tape sorts. A clear indication of the advantage of Sort III over both Sorts I and II can be calculated using data in Tables I and II. For every tenfold increase in elements to be sorted, there is a seventyfold increase in sort time using I and II, but only a fifteenfold increase using Sort III.

---

* This is another classic programming problem, the Sum-of-digits. Most teachers force their students to program the brute force sum to teach looping techniques rather than Gauss' elegant Sum = $(N^2+N)/2$.

FIGURE V--LOG-LOG TRANSFORMATION OF SORT TIME VS SORT SIZE

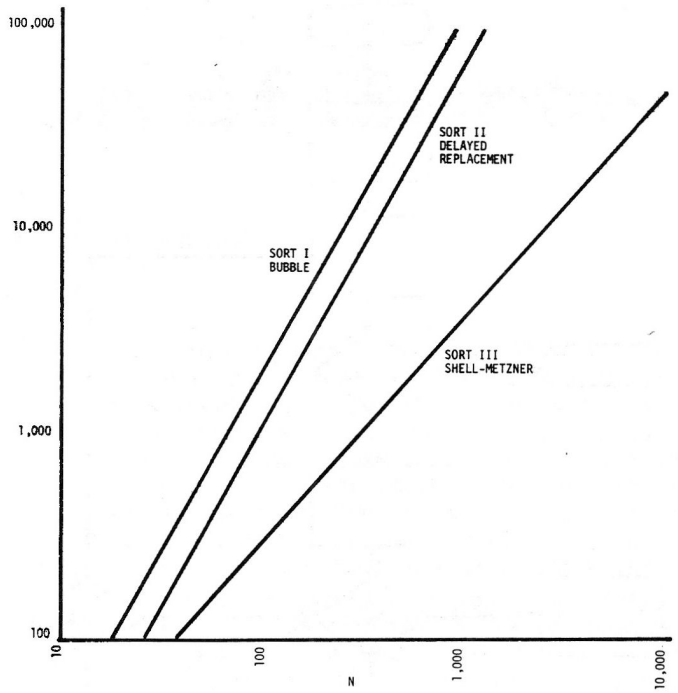


TABLE II--TIMING OF VERY LARGE CORE SORTS

N	Sort I	Sort II	Sort III
10,000	2.5 hrs	1.3 hrs	1 min
100,000	7.1 days	3.8 days	15 min
1,000,000	490 days	260 days	3.9 hrs
10,000,000	93 years	50 years	2.5 days

When this study was started, its purpose was to determine the crossover point at which the Shell-Metzner sort would begin to be more efficient than either of the other two. After all, it does take more coding space, and it does execute more statements given very small sorts. But after dealing with all three of these algorithms, it became more and more obvious that any production core sort code should use Sort III. The only excuse, weak as it is, for using either of the other two would be to teach the basics of sorting algorithms, or of following a flowchart. And under no circumstances should a student ever be taught the bubble sort or the delayed replacement sort without being presented the Shell-Metzner sort as well.

References:

1. Gottlieb, Byron S., Programming with BASIC, McGraw-Hill Book Company, Schaum's Outline Series in Accounting (1975), 175-183.
2. Stuart, Fredric, *FORTRAN Programming,* John Wiley and Sons (New York, 1969), 294-295. From a method published by Marlene Metzner, Pratt and Whitney Aircraft Company. From a method described by D. L. Shell.

John P. Grillo is a convert from chemistry to computing. He was an analytical chemist for Sandia Corporation in Albuquerque, New Mexico for four years. After three years of study at the University of New Mexico for his advanced degree, he taught at Albuquerque Technical-Vocational Institute, then the Computer Information Systems Department at West Texas State University. He is there now as an assistant professor. His research interests are the man-machine interface and CAI

```
10 DIM R2(2000)
20 PRINT "THIS PROGRAM PRODUCES A NORMALLY DISTRIBUTED SAMPLE"
30 PRINT "OF UP TO 2000 POSITIVE INTEGERS ACCORDING TO YOUR DEMANDS."
40 PRINT
50 PRINT "DO YOU WISH TO TIME SORTING ALGORITHMS";
60 INPUT S$
70 IF S$<>"YES" GOTO 150
80 PRINT "SELECT SORTING ALGORITHM:"
90 PRINT
100 PRINT "TYPE     TO USE"
110 PRINT " B        BUBBLE"
120 PRINT " R        DELAYED REPLACEMENT"
130 PRINT " S        SHELL - METZNER"
140 INPUT S$
150 PRINT "TYPE THE FOLLOWING:  SAMPLE SIZE, MEAN, STD. DEV."
160 S2=S4=F=0
170 X1=TIM
180 INPUT Y,M,S
190 IF Y<=2000 GO TO 240
200 PRINT "MAXIMUM SIZE = 2000"
210 GO TO 150
220
230
240 'COMPUTE RANDOM NOS. USING CENTRAL LIMIT THEOREM TECHNIQUE
250 FOR N=1 TO Y
260 R=0
270 FOR J=1 TO 12
280 R=R+RND
290 NEXT J
300 R=M+S*(R-6)
310 R2(N)=INT(R)
320 S2=S2+R2(N)
330 S4=S4+R2(N)*R2(N)
340 NEXT N
350 X2=TIM-X1
360 PRINT
370 PRINT
380 PRINT
390 PRINT Y"RANDOM NUMBERS GENERATED IN"X2"SECONDS."
400 PRINT
410 PRINT
420 M2=S2/Y
430 V2=S4-M2*S2
440 V2=V2/(Y-1)
450 PRINT "MEAN ="M2;
460 PRINT ", STD. DEV. ="SQR(V2)
470 PRINT
480 PRINT
490 PRINT "WHAT FORM OF OUTPUT DO YOU WANT?"
500 PRINT
510 PRINT "TYPE     IF YOU WANT"
520 PRINT " G       HISTOGRAM ON TTY"
530 PRINT " T       NUMBERS ON TTY"
540 PRINT " F       NUMBERS ON FILE"
550 PRINT " TS      NUMBERS ON TTY, SORTED"
560 PRINT " FS      NUMBERS ON FILE, SORTED"
570 INPUT O$
580 IF LEFT$(O$,1)<>"F" GOTO 630
590 PRINT "WHAT IS THE NAME OF THE FILE";
600 INPUT F$
610 FILE #1, F$
620 SCRATCH #1
630 IF O$<>"G" GOTO 680
640 IF F=1 GOTO 660
650 GOSUB 1230
660 GOSUB 1040
670 GOTO 930
680 IF RIGHT$(O$,1)<>"S" GOTO 710
690 IF F=1 GOTO 710
700 GOSUB 1230
710 IF LEFT$(O$,1)<>"F" GOTO 800
720 FOR A=1 TO Y BY 10
730 FOR B=A TO A+9
740 IF B>Y GO TO 930
750 PRINT #1,R2(B);
760 NEXT B
770 PRINT #1
780 NEXT A
790 GOTO 930
800 IF LEFT$(O$,1)="T" GOTO 850
810 IF LEN(O$)>0 GOTO 830
820 STOP
830 PRINT "IMPROPER OUTPUT CODE; TRY AGAIN"
840 GOTO 470
850 FOR A=1 TO Y BY 10
860 FOR B=A TO A+9
870 IF B>Y GOTO 920
880 PRINT R2(B);
890 NEXT B
900 PRINT
910 NEXT A
920 PRINT
930 PRINT "DIFFERENT OUTPUT";
940 INPUT O$
950 IF O$ = "YES" GOTO 510
960 IF O$<>"NO" GOTO 580
970 PRINT "DO YOU WANT ANOTHER SET OF NUMBERS";
980 INPUT T$
990 IF T$="YES" GO TO 150
1000 STOP
1010
1020
1030 'GRAPHING ROUTINE
1040 L=R2(1)
1050 H=R2(Y)
1060 I=(H-L)/30
1070 PRINT "GRAPH OF"Y"NUMBERS PRODUCED, FROM"L"TO"H"BY"I
1080 PRINT
1090 PRINT
1100 B=1
1110 FOR A=L TO H BY I
1120 PRINT INT(A),
1130 IF R2(B)<R2(B-1) GOTO 1200
1140 IF R2(B)>A GOTO 1180
1150 PRINT "*";
1160 B=B+1
1170 GOTO 1130
1180 PRINT
1190 NEXT A
1200 PRINT
1210 PRINT
1220 RETURN
1230 'SORTING ROUTINE
1240 F=1
1250 X1=TIM
1260 IF S$="R" GOTO 1500
1270 IF S$="B" GOTO 1320
1280 GOTO 1710
1290
1300
1310 'BUBBLE SORT
1320 PRINT "BUBBLE SORT ALGORITHM:"
1330 N7=C7=0
1340 FOR A=1 TO Y-1
1350 FOR B=A+1 TO Y
1360 C7=C7+1
1370 IF R2(A)<R2(B)GOTO 1420
1380 N7=N7+1
1390 T=R2(A)
1400 R2(A)=R2(B)
1410 R2(B)=T
1420 NEXT B,A
1430 X2=TIM-X1
1440 PRINT X2"SECONDS SORTING TIME."
1450 PRINT N7"SWITCHES EXECUTED."
1460 PRINT C7"COMPARISONS EXECUTED."
1470 RETURN
1480
1490
1500     'DELAYED REPLACEMENT SORT
1510     PRINT "DELAYED REPLACEMENT SORT ALGORITHM:"
1520 N7=C7=0
1530 J7=K7=L7=0
1540 L7=L7+1
1550 IF L7=Y GOTO 1430
1560 J7=L7
1570 K7=J7+1
1580 C7=C7+1
1590 IF R2(K7)>R2(J7) GOTO 1610
1600 J7=K7
1610 K7=K7+1
1620 IF K7<=Y GOTO 1580
1630 IF L7=J7 GOTO 1540
1640 N7=N7+1
1650 T=R2(J7)
1660 R2(J7)=R2(L7)
1670 R2(L7)=T
1680 GOTO 1540
1690
1700
1710 'SHELL - METZNER SORT
1720 PRINT "SHELL - METZNER SORT:"
1730 N7=C7=0
1740 M6=Y
1750 M6=INT(M6/2)
1760 IF M6=0 GOTO 1430
1770 K6=Y-M6
1780 J6=1
1790 I6=J6
1800 L6=I6+M6
1810 C7=C7+1
1820 IF R2(I6)<=R2(L6) GOTO 1890
1830 N7=N7+1
1840 T=R2(I6)
1850 R2(I6)=R2(L6)
1860 R2(L6)=T
1870 I6=I6-M6
1880 IF I6>=1 GOTO 1800
1890 J6=J6+1
1900 IF J6>K6 GOTO 1750
1910 GOTO 1790
1920 END

READY
```

PROGRAM TO TEST SORT ALGORITHMS

```
THIS PROGRAM PRODUCES A NORMALLY DISTRIBUTED SAMPLE
OF UP TO 2000 POSITIVE INTEGERS ACCORDING TO YOUR DEMANDS.

DO YOU WISH TO TIME SORTING ALGORITHMS ?YES
SELECT SORTING ALGORITHM:

TYPE      TO USE
 B         BUBBLE
 R         DELAYED REPLACEMENT
 S         SHELL - METZNER
 ?R
TYPE THE FOLLOWING:  SAMPLE SIZE, MEAN, STD. DEV.
 ?100,100,15


 100 RANDOM NUMBERS GENERATED IN 0.316 SECONDS.


MEAN = 100.81 , STD. DEV. = 15.5879


WHAT FORM OF OUTPUT DO YOU WANT?

TYPE     IF YOU WANT
 G       HISTOGRAM ON TTY
 T       NUMBERS ON TTY
 F       NUMBERS ON FILE
 TS      NUMBERS ON TTY, SORTED
 FS      NUMBERS ON FILE, SORTED
 ?T
 101  149   98   88   69   81  104  115  131  111
 115  105  101   82  120  113   88  107  108  115
 95  117   80   89  111   97  115   95  105  103
 109  102   84  127   99  113   96   77   98   93
 87   97   74  125  114  135   83  103   90  100
 108  104   93   93  112  106   85   76  112  123
 103  110   73   88  104   91  120  107  133  106
 105   80  106   76  110   82   82   83   92   92
 100  102  121  103   72  101  108   88  114  119
 120  111   71   89   95  106   94   88   90  120
```

```
DIFFERENT OUTPUT ?TS
DELAYED REPLACEMENT SORT ALGORITHM:
 0.967 SECONDS SORTING TIME.
 96 SWITCHES EXECUTED.
 4950 COMPARISONS EXECUTED.
 69   71   72   73   74   76   76   77   80   80
 81   82   82   82   83   83   84   85   87   88
 88   88   88   88   89   89   90   90   91   92
 92   93   93   93   94   95   95   95   96   97
 97   98   98   99  100  100  101  101  101  102
 102  103  103  103  103  104  104  104  105  105
 105  106  106  106  106  107  107  108  108  108
 109  110  110  111  111  111  112  112  113  113
 114  114  115  115  115  115  117  119  120  120
 120  120  121  123  125  127  131  133  135  149

DIFFERENT OUTPUT ?G
GRAPH OF 100 NUMBERS PRODUCED, FROM 69 TO 149 BY 2.66667


 69           *
 71           *
 74           ***
 77           ***
 79
 82           ******
 85           ****
 87           *
 90           *********
 93           ******
 95           ****
 98           *****
101           ******
103           ******
106           **********
109           ******
111           *****
114           ******
117           *****
119           *
122           *****
125           **
127           *
130
133           **
135           *
138
141
143
146
```

The Systems Approach:

How to evaluate, design, and implement a software application

by John R. Lees

Have you ever thought up an application, you know, "Gee, I wish I had a program to dindle my framistan," and started to write it, maybe get a little code running, but bog down somewhere and never carry through? If so, it's quite possible that you were suffering from a lack of systems approach. Perhaps the single most important step in completing a software project, and the one most often neglected by the big and little programmer alike, the systems approach consists of thinking things through in advance.

Sure, that sounds simple and obvious ("I thought things through, I want a program to dindle my framistan"). However, it *isn't* simple and obvious. Large "real world" programming projects spend a significant portion of their time and budgets in coming up with a system design. Of course for your own personal project you're not going to be worried about things like how many programmers you can effectively use during each phase of the project, and whether you need a project librarian, but there are a number of techniques that have been developed that will be of benefit to you.

I. Iterative procedure of refinement and repetition. The final result is, hopefully, a project which will work.

II. Hardest thing to do is to get a good overall picture of what you want before you have it.

A. Think it through; try to imagine using your completed application. Try to make a list of everything you want to be able to do and how you want to do it.

B. If others are going to use the application, get their input. Good idea to talk it over with someone else, anyway.

III. If the project is large, break it up into parts which can be coded and tested separately.

IV. Be realistic in evaluating storage, time, interface requirements. Remember you have a small system and may have to make sacrifices in your design to get it implemented.

IV. Plan files, storage, subroutines, etc.

V. Once you get a design, STICK WITH IT! Do not give in to the temptation to change things in midstream. That is the single most prevalent reason for projects never being completed.*

VI. Figure out how to test it before using it. ■

*A recent Rand Corp. study (read thorough and costly) indicates that the ratio of the actual time to complete a well-planned project compared to the estimated time is 3 to 1.

Heapsort

Most programming texts present the problem of writing one or two basic types of sort programs. Are these generally used in production? Usually not. One of the most efficient production sort algorithms is known as *Heapsort.* In the richly commented BASIC program below, Geoffrey Chase, OSB, of the Portsmouth Abbey School has written a *Heapsort* routine for both character string or numeric sorting. Look it over. Study how it works. And when you want a really efficient sort routine, use it!

```
NOTES:
(1) EVIDENTLY THIS CAN BE SPLIT INTO TWO PROGRAMS; OR YOU
    CAN CUT OUT THE UNNEEDED HALF.

(2) LINE 120 CAN BE DIMENSIONED AS DESIRED.

(3) THE  !  "TAG" COMMENTS AREN'T NEEDED.  SOME BASICS
    ALLOW  !  , SOME ALLOW  '  INSTEAD, SOME NEITHER.
```

```
100 REM.  KNUTH/WILLIAMS/FLOYD  HEAPSORT ALGORITHM.
110 !   PAS '74
120 DIM N(150),C$(150)
130 PRINT
135 PRINT
140 PRINT
145 PRINT "TYPE  C  FOR CHARACTER STRING SORT,"
150 PRINT "TYPE  N  FOR NUMBER SORT. ";
155 INPUT W$
160 N=0                             ! START COUNT=N AT 0
163 PRINT
166 PRINT
170 IF W$="N" THEN 480
180 IF W$<>"C" THEN 140             ! BAD REPLY
190 !--------------------------< CHARACTER SORT: >----
200 GOSUB 720                       ! ASK FOR STOP CODE
210 INPUT S$                        ! GET STOP CODE
215 PRINT
220                                 ! INPUT LOOP:
230 N=N+1
235 INPUT C$(N)
240 IF C$(N)<>S$ THEN 230
250                                 ! END OF INPUT...
260 N=N-1
265 PRINT
270                                 ! HEAPSORT PROPER:
280 L=INT(N/2)+1
285 N1=N                            ! PRESERVE N, USE N1
290 IF L=1 THEN 310
300 L=L-1
303 A$=C$(L)
306 GOTO 350
310 A$=C$(N1)
315 C$(N1)=C$(1)                    ! MOVE TOP OF HEAP TO END
320 N1=N1-1                         ! HEAP IS 1 SMALLER NOW
330 IF N1=1 THEN 440                ! ONLY ONE LEFT? THEN WE!RE DONE.
340                                 ! NO, CONTINUE
350 J=L
360 I=J
365 J=2*J                           ! LOOK FOR "SONS" OF I
370 IF J=N1 THEN 400
380 IF J>N1 THEN 420                ! "N1" IS SIZE OF ACTIVE LIST
390 IF C$(J)>=C$(J+1) THEN 400      ! CHOOSE LARGER "SON"
395 J=J+1
400 IF A$>=C$(J) THEN 420
410 C$(I)=C$(J)
415 GOTO 360                        ! LARGER SON REPLACES PARENT
420 C$(I)=A$
425 GOTO 290
430                                 ! END OF SORT...
440 C$(1)=A$
450 FOR I=1 TO N
453 PRINT C$(I)                     ! OR REVERSE ORDER: I=N TO 1 STEP -1
456 NEXT I
460 GOTO 130
470 !--------------------------< NUMERIC SORT: >-----
480 GOSUB 720
483 INPUT S
486 PRINT
490 N=N+1
493 INPUT N(N)
496 IF N(N)<>S THEN 490
500 N=N-1
505 PRINT
510     !
520 L=INT(N/2)+1
525 N1=N
530 IF L=1 THEN 550
540 L=L-1
543 A=N(L)
546 GOTO 590
550 A=N(N1)
555 N(N1)=N(1)
560 N1=N1-1
570 IF N1=1 THEN 680
580     !
590 J=L
600 I=J
605 J=2*J
610 IF J=N1 THEN 640
620 IF J>N1 THEN 660
630 IF N(J)<N(J+1) THEN J=J+1    ! FANCY "IF" SYNTAX. COMPARE
640 IF A>N(J) THEN 660                                  390-400.
650 N(I)=N(J)
655 GOTO 600
660 N(I)=A
665 GOTO 530
670     !
680 N(1)=A
690 FOR I=1 TO N
693 PRINT N(I)
696 NEXT I
700 GOTO 130
710 !--------------------------< SUBROUTINE: >-------
720 PRINT "PLEASE INDICATE A STOP CODE--SOMETHING NOT IN YOUR"
730 PRINT "LIST, WHICH WILL ACT AS AN 'END-OF-LIST' SIGNAL: ";
740 RETURN
750     !
760 END
```

```
      TYPE  C  FOR CHARACTER STRING SORT,
      TYPE  N  FOR NUMBER SORT. ? C


      PLEASE INDICATE A STOP CODE--SOMETHING NOT IN YOUR
      LIST, WHICH WILL ACT AS AN 'END-OF-LIST' SIGNAL: ? KNUTH

      ? DAVID AHL, ESQ.
      ? COSMO COMPUTERS
      ? ABPLANALP LTD.
      ? PETRODOLLARS
      ? DMA TRANSFER
      ? CREATIVE COMP.
      ? M.O.S. ABACUS
      ? ALGORITHMS
      ? LEONARDO P.
      ? CHINESE REMS.
      ? SORTED STRINGS
      ? NEG. FULLBACK
      ? STAR TREK, V.2
      ? KNUTH

      ABPLANALP LTD.
      ALGORITHMS
      CHINESE REMS.
      COSMO COMPUTERS
      CREATIVE COMP.
      DAVID AHL
      DMA TRANSFER
      LEONARDO P.
      M.O.S. ABACUS
      NEG. FULLBACK
      PETRODOLLARS
      SORTED STRINGS
      STAR TREK


      TYPE  C  FOR CHARACTER STRING SORT,
      TYPE  N  FOR NUMBER SORT. ? N


      PLEASE INDICATE A STOP CODE--SOMETHING NOT IN YOUR
      LIST, WHICH WILL ACT AS AN 'END-OF-LIST' SIGNAL: ? -1E6

      ? 3.1416
      ? 22222
      ? 2E10
      ? 2E-10
      ? 66.666
      ? -1E5
      ? -1E6

      -100000
       2.00000E-10
       3.1416
       66.666
       22222
       2.00000E+10


      TYPE  C  FOR CHARACTER STRING SORT,
      TYPE  N  FOR NUMBER SORT. ?
      STOP AT LINE 155
      READY
```

A New Fast Sorting Algorithm

How to sort extremely fast with a minimum of comparisons, and a minimum of programming steps between comparisons.

Richard Hart*

The speed or efficiency of an internal sorting algorithm is directly related to the number and complexity of programming steps which are actually executed during its operation. Because all the items being sorted must be compared to one another to place them in order, programmers have realized that by making certain intelligent comparisons, the total number of comparisons executed in an algorithm may be reduced to a theoretical minimum.

This article starts with the theory described by Luther J. Woodrum in the *IBM Systems Journal* and describes an algorithm that not only uses a minimal number of comparisons, but also executes a minimal number of programming steps between each comparison.

The algorithm was written to use as few comparisons as possible, to have as few steps between each comparison as possible, to take advantage of natural sequencing, to preserve the order of equals (or even the reverse order of equals), to avoid moving records around, to use as little memory as possible (one working array), and to be a modular, easily understood program written in BASIC.

29 Concord Ave., Apt. 609, Cambridge, MA 02138

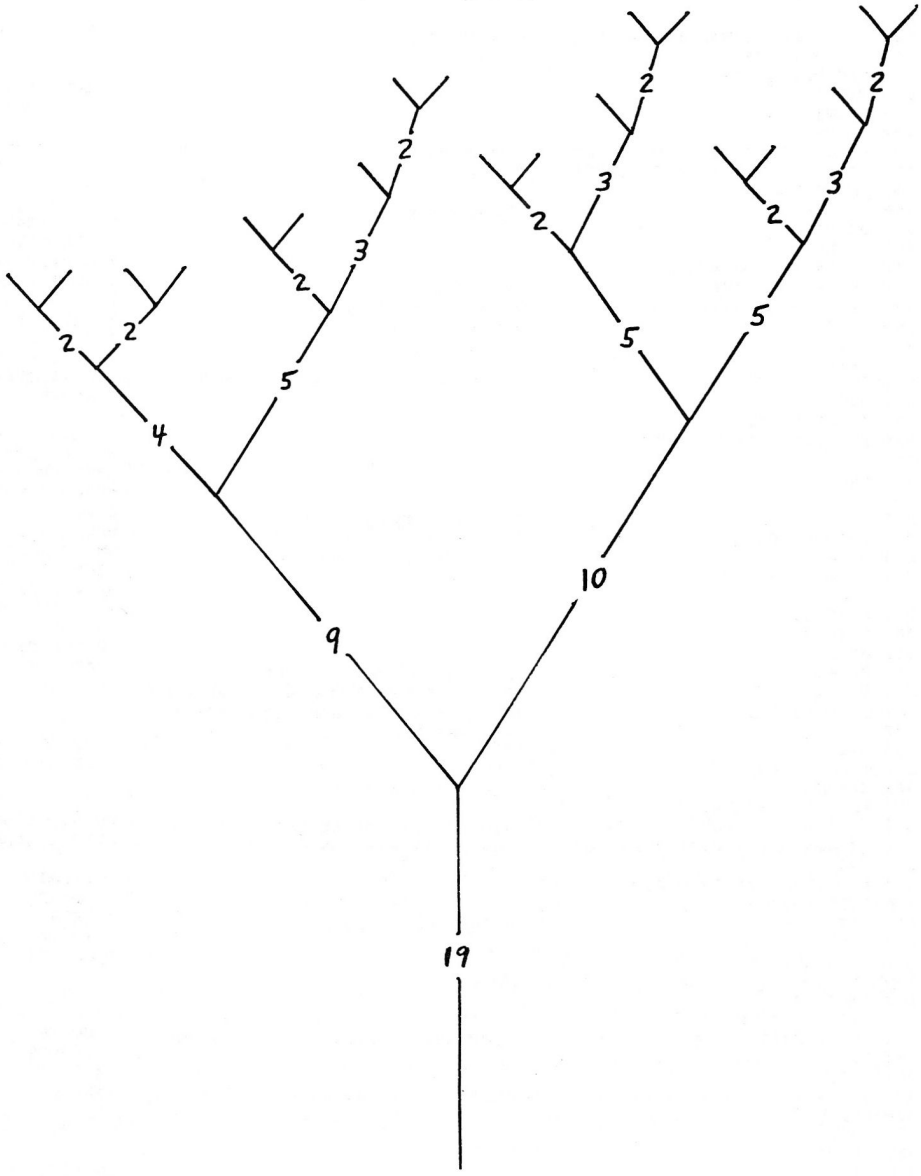
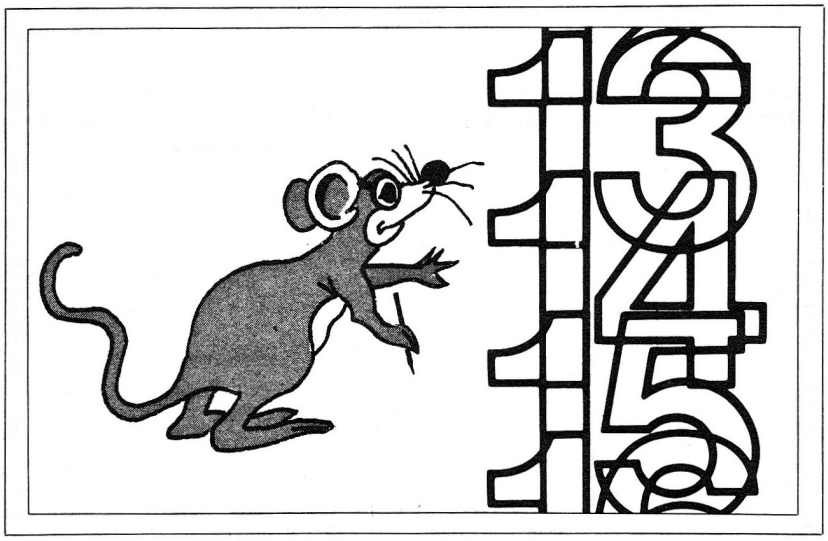


Fig. 1. Tree number 19, with 19 leaves and nearly equal branches.



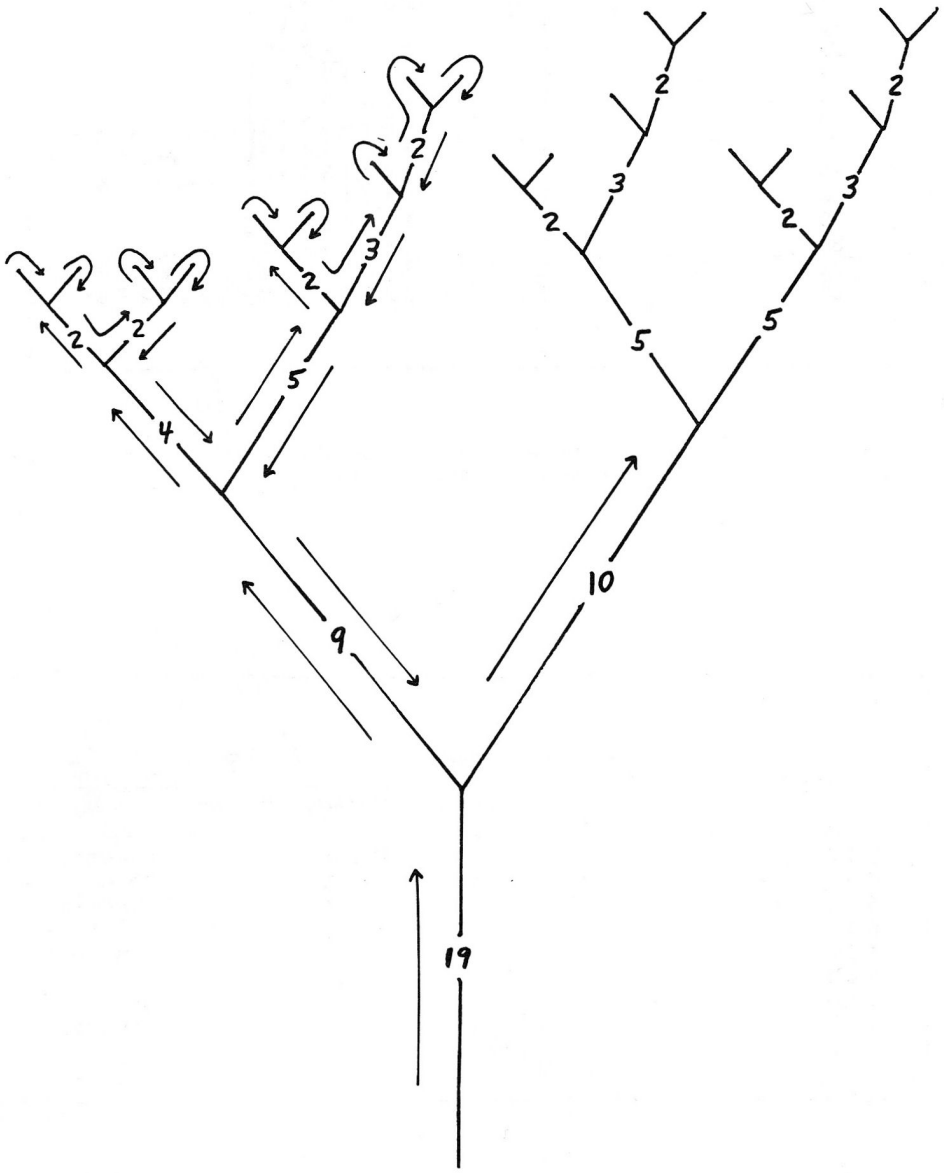


Fig. 2. Woodrum's algorithm for sorting N records with minimal comparing: follow the tree structure and merge nearly equal linked lists on the way down the branches.

The Forest.

The theory behind the algorithm may be described in a language of forests, trees, branches, twigs and leaves. There is a forest filled with trees of different sizes. The smallest tree has one leaf and the largest has no more leaves than fit into the memory of a computer. Each tree is very orderly. The trunk of a tree splits into two branches of nearly the same size. If one branch is larger than the other, it is always the right-hand branch. Similarly, each branch divides into two more branches until the branches become twigs from which leaves grow. Figure 1 is a picture of tree number 19, which has 19 leaves.

Each tree in the forest is almost a binary system of its own, but the leaves are wildly disarrayed. They have different colors, different shapes, different sizes, different ages and different fates. Luckily the leaves of any particular tree, even the largest, can fit into the memory of a computer where they can be arranged by color, by texture, or by intelligence, and shipped to any part of the world.

The Mouse.

Even though the computer usually thought about everything in two ways, it worked well in the forest. It had evolved from the Boolean logic of electrical engineering into a tiny, nearsighted mouse. She could do anything. She could climb the trees, jump from twig to twig; she could think like a calculator and even read, one letter at a time.

If someone in the outside world thought that the leaves of a particular tree were ripe and wanted them arranged by taste (from sweet to sour to bitter) and mailed to his home, the mouse would begin her work. The mouse liked the forest. Most people only saw the wildly disarranged leaves on the outside, but on the inside, the trees and leaves were arranged perfectly for the mouse.

One day the mouse was idling by the edge of the forest, thinking yes and no, when a secretary from IMOK telephoned to say that he would like the leaves from tree 19 arranged alphabetically by color, xeroxed and sent to his office. He mentioned that each leaf's exact color was printed on the leaf exactly two inches from the tip. This was especially helpful information for the mouse because even though she was a whiz with digits and knew her ABC's, she was color-blind.

The mouse began to work. She calculated her course and started along toward tree 19. She was thinking nothing at all.

The Spider.

Deep in the forest, at the entrance to a cave, was a spider who was by profession a computer programmer. Strange as it may seem, he hardly ever saw the mouse, but by looking deep into the cave he would imagine where the mouse might be. Occasionally he would turn around and glimpse the forest. In a single instant he would imagine it was his own forest, that the binary trees were his as was the mouse. Then he would turn back into his cave and write computer programs.

One day the spider read a modest article by Luther J. Woodrum in the *IBM Systems Journal,* Vol. 8, No. 3, 1969, called "Internal sorting with minimal comparing." The Woodrum algorithm was written in APL, a foreign language, but the spider liked sorts so he translated the algorithm into BASIC, his native language.

Then he noticed two surprising things: The Woodrum sort was faster than the Shell sort and was also the same procedure he had seen the mouse go through time after time in the forest.

The mouse would start on the left-hand side of a tree and climb until she reached the lowest branch; she would climb out that branch until she reached the left-most leaf. Then she would climb from leaf to leaf, looking at each one and writing something on an adding-machine tape. So the mouse was creating linked lists and merging them! The trees here were perfect for balanced merges! Every time the mouse climbed up a branch she would be figuring out what to do. Then on the way down to the next branch she would be merging the leaves and branches she had left behind. Figure 2 shows the path the mouse was following to sort N records (19 in this case) with a minimum number of comparisons.

With a new understanding the spider watched the mouse for three years climbing the different trees, setting up linked lists of length 1 at each leaf, and allowing the tree structure to create balance merges. It had occurred to the spider in the beginning that it would be easier for the mouse to jump from leaf to leaf rather than to follow the limbs around. He had mentioned this to the mouse and the mouse had tried it, gamely enough. But the mouse couldn't see the tree structure below so she had ended up with 19 linked lists of length 1 and none of them merged.

Tree number	*Number of low-order twigs*	*Twig list*	
1		EXIT - - - -	
2	1	2	
3	1	3	
4	2	2 2	
5	1	2 3	
6	2	3 3	
7	1	3 4	
8	4	2 2 2 2	
9	3	2 2 2 3	
10	2	2 3 2 3	
11	1	2 3 3 3	
12	4	3 3 3 3	
13	3	3 3 3 4	
14	2	3 4 3 4	
15	1	3 4 4 4	
16	8	2 2 2 2 2 2 2 2	2-leaf twigs and 3-leaf twigs
17	7	2 2 2 2 2 2 2 3	
18	6	2 2 2 3 2 2 2 3	
19	5	2 2 2 3 2 3 2 3	
20	4	2 3 2 3 2 3 2 3	
21	3	2 3 2 3 2 3 3 3	
22	2	2 3 3 3 2 3 3 3	
23	1	2 3 3 3 3 3 3 3	
24	8	3 3 3 3 3 3 3 3	3-leaf twigs and 4-leaf twigs
25	7	3 3 3 3 3 3 3 4	
26	6	3 3 3 4 3 3 3 4	
27	5	3 3 3 4 3 4 3 4	
28	4	3 4 3 4 3 4 3 4	
29	3	3 4 3 4 3 4 4 4	
30	2	3 4 4 4 3 4 4 4	
31	1	3 4 4 4 4 4 4 4	
32	16	2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2	
33	15	2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3	

Fig. 3. Trees are composed of 2-leaf and 3-leaf twigs or else 3-leaf and 4-leaf twigs. Within one of these tree groups, new high-order twigs appear in a reflected binary pattern.

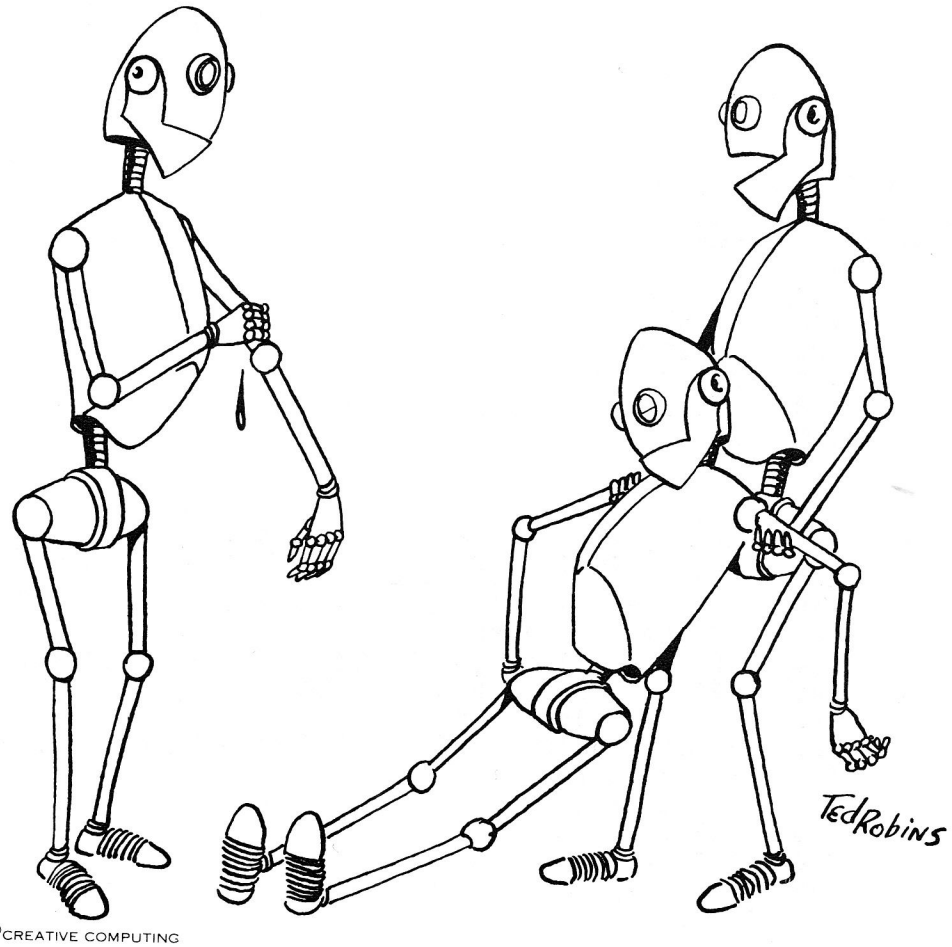
Counter 1 2 4	*Mirror image value*	*Number of carries*	*Tree 19*		
			Generate	*and*	*Merge*
0 0 0	0		2-twig		0 branches
0 0 1	4	0	2-twig		1 branch
0 1 0	2	1	2-twig		0 branches
0 1 1	6	0	3-twig		2 branches
1 0 0	1	2	2-twig		0 branches
1 0 1	5	0	3-twig		1 branch
1 1 0	3	1	2-twig		0 branches
1 1 1	7	0	3-twig		3 branches
		3			

Fig. 4. The binary counter provides all the necessary information for merging. The reflected value determines how many new leaves to generate and the tally of carries indicates the number of additional branches to merge.
This table shows the correspondence between a reflected binary counter and the sequential merging that occurs in tree 19. The cutoff value for tree 19 is 5 (from Fig. 3). If the "mirror-image" counter is below 5, generate a 2-twig; otherwise generate a 3-twig.

		Array L					
Position	Unsorted records	Before 1st merge	Before 2nd merge	Before 3rd merge	Before 4th merge	Before last merge	Before last merge
1	violet	1*	1	1	1	1	18
2	indigo	2*	1	1	3	3	15
3	orange		3*	4	4	6	6
4	purple		4*	4	1	5	5
5	red				5*	1	16
6	pink				6*	4	4
7	blue					8	8
8	brown					9	11
9	green					2	10
10	grey					15	2
11	carmine					17	17
12	yellow					12	12
13	olive					16	3
14	black					11	7
15	infrared					13	13
16	rust					18	1
17	crimson					10	9
18	white					12	12
19	amber					14	14
(20)		1*	2	2	2	7	19
(21)		2*	3*	3	5*	19	
(22)			4*		6*		
(23)							
(24)							

The starred sequences are newly generated lists one record long (new leaves).

Fig. 5. Snapshots of linked lists during execution of the algorithm. The last column shows the final linked list that begins at position N+1 in array I: (20) 19 14 7 8 11 ... Each location shows the value of the next location except that the last location points to itself.



"He can't stand the sight of oil."

The spider busied himself with other things: he visited other forests with fibonacci trees and even pure binary trees, but most of these trees had an extra branch sprouting from the side to hold leftover leaves. These trees weren't so good for sorting. His own mouse always came closest to sorting with a theoretical minimum number of comparisons.

The Insight.

One day the spider glanced out of his cave long enough to see something curious about the trees in his forest. He didn't know exactly what he saw so he asked the mouse to give him a twig list for trees 1 through 33. (A twig is composed of 2, 3 or 4 leaves). He wanted to see all the twigs at the exact altitude where the leftmost twig was less than 4 leaves; see Figure 3.

The spider noticed that this left-most twig was always a low-order twig and that the total number of twigs at that altitude was always an even binary number. There were two fundamentally different kinds of trees: those with 2 and 3 twigs and those with 3 and 4 twigs.

Now as the spider looked from one tree to the next higher tree (look at trees 16 through 23), he noticed that the high-order twigs sprouted first from each half of the tree, then from each remaining quarter and so on, until all but the first twig was a high-order twig. The spider immediately realized he could use this binary pattern to help the mouse.

At just this moment, the mouse was on her way to tree 19 to arrange the leaves for the secretary from IMOK.

The spider picked up a mirror and met the mouse at the tree. Then the spider told the mouse what to do:

1. Before you climb the tree, calculate what the low-order twig will be. (For tree 19, it's a 2-twig, the first left-most twig containing fewer than four leaves).
2. Calculate how many of these twigs will be at that altitude. (5).
3. Calculate the total number of twigs at that altitude; this total will always be an even binary number and will determine the size of the binary counter in the next step. (8).
4. Take a binary counter that counts from 0 to 7 and this mirror. Climb the tree to the left-most twig and set the counter to 0. Then proceed to leap from twig to twig and increase your counter by 1 each time you leap. *Look at the counter in the mirror* and if that mirror-image value is less than the number of low-order twigs (5), create linked lists for a 2-leaf twig; otherwise create linked lists for a 3-leaf twig.

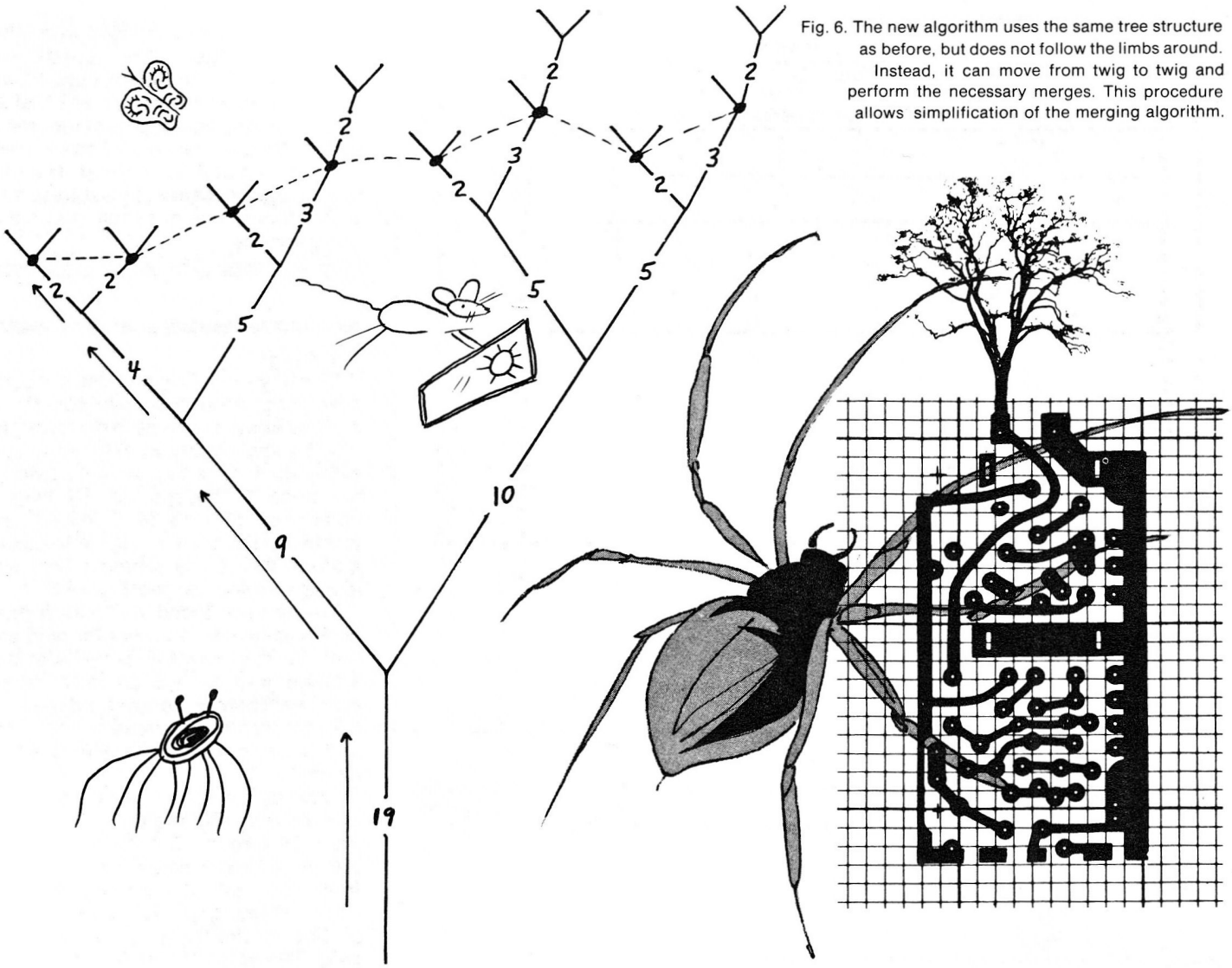


Fig. 6. The new algorithm uses the same tree structure as before, but does not follow the limbs around. Instead, it can move from twig to twig and perform the necessary merges. This procedure allows simplification of the merging algorithm.

The Accident.

So once again the mouse followed the spider's instructions; she leaped from twig to twig and wrote on an adding-machine tape. Then the mouse climbed down the right side of the tree and showed the spider what had happened. The leaves had all been merged into 8 twigs and the binary counter had provided some more unexpected information!

The mouse had noticed that the counter had clicked every time it carried a digit. The number of these clicks corresponded exactly to the number of merges needed below the previous twig to merge twigs or branches into larger branches. Now the mouse could easily figure exactly how many leaves were above the twig and how many branches were below the twig without leaving the twig! See Figure 4.

So the spider changed his algorithm to use the previous twig, combined all the working arrays into one, and created a beautiful butterfly merge to combine leaves into twigs, twigs into branches and branches into one final linked list starting at position N+1. In the end L(N+1) points to the first leaf;

Fig. 7. This is the new sorting algorithm. Array L must have room for N+LOG2(N)+2 elements. The algorithm uses a minimal number of comparisons and a minimal number of steps between each comparison.

```
100   DIM N(1000)
110   PRINT "SORT HOW MANY RANDOM NUMBERS";
120   INPUT N
130   FOR I = 1 TO N
140   LET N(I) = INT(RND(0)*10000+1)
150   NEXT I
160   !
170   !
180   REM  ENTRY
190   DIM L(1011)   !LINKS:  N+LOG2(N)+2 ELEMENTS
200   LET K1,I,M1,T2,T4 = 0
210   LET J = N+1   !HEAD OF SEQUENCE 1
220   LET L(1),L(J),K2 = 1
230   IF N <= 1 THEN 940   !EXIT;  NOTHING TO SORT
240   LET S1 = N   !NUMBER OF LEAVES
250 REM  CLIMB THE TREE
260   IF S1 < 4 THEN 320   !LOW-ORDER TWIG VALUE
270   LET K2 = K2*2   !TOTAL NUMBER OF TWIGS
280   LET B2 = S1/2
290   LET S1 = INT(B2)
300   LET T4 = T4+(B2-S1)*K2
310   GO TO 250
320 REM  INITIAL CALCULATIONS
330   LET T4 = K2-T4   !NUMBER OF LOW-ORDER TWIGS
```

```
340   LET B2 = K2/2   !HIGH BIT VALUE OF BINARY COUNTER
350 REM  NEXT TWIG
360   IF K1 = K2 THEN 940   !EXIT;  SORT COMPLETE
370   LET K1,T1 = K1+1   !TWIG NUMBER
380   LET B1 = B2   !HIGH BIT VALUE
390   LET T3 = T2   !PREVIOUS REFLECTED TWIG NUMBER
400 REM  ADD 1 TO REFLECTED BINARY COUNTER AND CARRY
410   LET T1 = T1/2
420   IF INT(T1) < T1 THEN 470   !NO MORE CARRIES
430   LET M1 = M1+1   !NUMBER OF MERGES
440   LET T2 = T2-B1
450   LET B1 = B1/2   !NEXT BIT VALUE
460   GO TO 400   !CARRY ONE
470 REM  TWIG CALCULATIONS
480   LET T2 = T2+B1   !REFLECTED TWIG NUMBER
490   IF S1 = 2 THEN 550   !2-TWIGS AND 3-TWIGS
500   REM  3-TWIGS AND 4-TWIGS
510   IF T3 < T4 THEN 560   !LOW-ORDER TWIG (3-TWIG)
520   REM  4-TWIG
530   LET M1 = -M1   !DIS-ENGAGE NUMBER OF MERGES
540   GO TO 630
550 IF T3 < T4 THEN 610   !LOW-ORDER TWIG (2-TWIG)
560 REM  3-TWIG
570   LET M1 = M1+1   !NUMBER OF MERGES
580   LET I = I+1   !NEXT LEAF
590   LET L(I),L(J) = I   !GENERATE A LEAF
600   LET J = J+1   !NEXT SEQUENCE HEAD
610 REM  2-TWIG
620   LET M1 = M1+1   !NUMBER OF MERGES
630 LET I = I+1   !NEXT LEAF
640   LET L1,L(I),L(J) = I   !GENERATE A LEAF
650   LET L0 = J   !HEAD OF OLDER LEAF (LAST LINE)
660   LET J = J+1   !HEAD OF LATEST LEAF (NEXT 2 LINES)
670   LET I = I+1   !NEXT LEAF
680   LET L2,L(I),L(J) = I   !GENERATE A LEAF
690   GO TO 750   !MERGE LEAVES
700 REM  MERGE TWIGS AND BRANCHES
710   LET J = J-1   !HEAD OF LATEST BRANCH OR TWIG
720   LET L0 = J-1   !HEAD OF OLDER BRANCH OR TWIG
730   LET L1 = L(L0)   !HEAD OF SEQUENCE 1
740   LET L2 = L(J)   !HEAD OF SEQUENCE 2
750 IF N(L1) <= N(L2) THEN 820   !STAY IN SEQUENCE 1
760   LET L(L0) = L2   !SWITCH TO SEQUENCE 2
770 LET L0 = L2   !TOP LEAF IN SEQUENCE 2
780   LET L2 = L(L0)   !NEXT LEAF IN SEQUENCE 2
790   IF L2 = L0 THEN 870   !END OF SEQUENCE 2
800   IF N(L1) > N(L2) THEN 770   !STAY IN SEQUENCE 2
810   LET L(L0) = L1   !SWITCH TO SEQUENCE 1
820 LET L0 = L1   !TOP LEAF IN SEQUENCE 1
830   LET L1 = L(L0)   !NEXT LEAF IN SEQUENCE 1
840   IF L1 <> L0 THEN 750   !NOT END OF SEQUENCE 1
850   LET L(L0) = L2   !SWITCH TO SEQUENCE 2
860   GO TO 880
870 LET L(L0) = L1   !SWITCH TO SEQUENCE 1
880 LET M1 = M1-1   !NUMBER OF MERGES
890   IF M1 > 0 THEN 700
900   IF M1 = 0 THEN 350
910   REM  GENERATE 2ND HALF OF A 4-TWIG
920   LET M1 = 1-M1   !RE-ENGAGE NUMBER OF MERGES
930   GO TO 630
940 REM  EXIT
950   LET L0 = N+1   !FIRST LINK IN SEQUENCE
960   !
970   !
1000   FOR I = 1 TO N
1010   LET L0 = L(L0)   !FOLLOW LINKS
1020   PRINT N(L0);
1030   NEXT I
1050   END
```

L(L(N+1)) points to the second, L(L(L(N+1))) points to the third, and the last link points to itself. See Figure 5.

The Butterfly Merge.

Two things happen as the mouse jumps from twig to twig. The leaves above the mouse get merged into the twig and the twigs and branches behind the mouse get merged into larger branches. The butterfly merge treats each merge the same way. The heads of each sequence are kept at positions N+1, N+2, . . . , N+INT(LTW(N)+2) after the links themselves, which are kept in positions 1, 2, . . . , N of array L.

The merge takes the last two sequences in the list and combines them into one. One wing of the merge follows sequence 1 and the other follows sequence 2. The two are interwoven until the final link points to itself. Because the heads of each sequence are kept in the same array with the links themselves, the merge is extraordinarily fast. After each merge, the stack of sequence heads has been reduced by one.

Generating Leaves.

Each time the mouse reaches a new twig, she generates new sequences one item long to correspond to the leaves of that twig. A two-leaf twig is produced by creating two one-item sequences, each pointing to itself. Then these two leaves are merged once. A three-leaf twig is created from three one-item sequences merged twice. A four-leaf twig is merged from two two-leaf twigs: The first two-leaf twig is generated and merged once; then the number of remaining merge passes is set to a negative number so that the merge will be disabled until the second two-leaf twig is generated and merged with the first.

After each complete twig has been generated, merging continues until the branches behind the mouse have been linked together. Then the mouse jumps to the next twig, generates new leaves and lets the butterfly merge fly by again.

Now the mouse follows this procedure all the time. After the spider watched the mouse a few times, he turned into his cave and forgot. But every now and then sunlight shines through the leaves of the forest, reflects from the mouse's mirror and flashes deep into his cave. See Figures 6 and 7. ■

Shuffling

by John Jaworski
Hatfield Polytechnic, England

When faced with the problem of printing out the integers from 1 to 10 in a random order (without repetitions), the following program is an excellent example of how not to proceed:

```
100  FOR I = 1 TO 10
110  N = INT(10*RND+1)
120  PRINT N;
130  NEXT I
140  END
```

As we are taking no precautions against repetitions, we can be almost certain to get some. In fact, if my rapid calculations are correct, the probability of getting what we are looking for is minute: 3.6×10^{-4} — rather less than 4 correct solutions in 10,000 trials!

Somewhat better, on the face of it, is the next example. Here we provide an array M, choose a random integer and only insert it into the array M when we have checked that this integer is not already present.

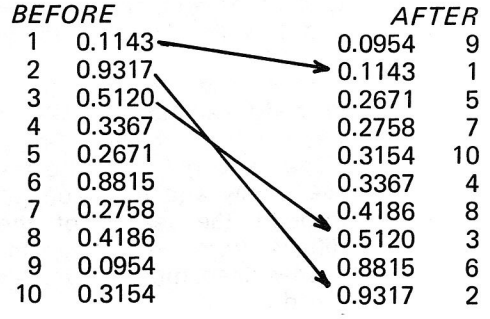
Note that statment 100 is superfluous in the BASIC language, but it is as well to remind ourselves of the nature of M.

```
100  DIM M(10)
110  K=1
120  N=INT(10*RND+1)
130  FOR J=1 TO K
140  IF M(J)=N THEN 120
150  NEXT J
160  M(K)=N
170  K=K+1
180  IF K<11 THEN 120
190  MAT PRINT M;
200  END
```

Unfortunately, while producing results, this is not at all economical on time. When the array is almost full, say with 9 numbers inserted, there is in fact no choice at all for the last element, but only one chance in ten that statement 120 will select the correct integer for us — 90% of the work done by the program at this stage is wasted. This of course is more acute when shuffling rather more integers!

One rather more elegant method is the following: choose ten random *real* numbers less than 1, using RND. Associate these with the ten integers 1 - 10 and then sort them into order, shifting the integers around at the same time:

e.g.



We can sort the real numbers by any sorting method we have found suitable. Here is a demonstration program that uses the 'ripple' sort (which you may know under some other name).

```
100  DIM A(10), P(10)          A contains the random
110  FOR I = 1 TO 10           numbers and P the inte-
120  P(I) = I                  gers that will eventually
130  A(I) = RND                be printed. This loop
140  NEXT I                    sets up these arrays.

150  L = 10                    The limit on compari-
160  F = 0                     sons, L is set to 10, the
170  I = 1                     flag is set and we begin
                               our comparisons at the
                               beginning of the list.

180  IF A(I) > A(I+1) THEN 260 compare adjacent
                               numbers.

190  T = A(I)                  if not in order, swap
200  A(I) = A(I+1)             A's . . .
210  A(I+1) = T

220  T = P(I)
230  P(I) = P(I+1)             . . . swap P's
240  P(I+1) = T

250  F = 1                     and set flag

260  I = I+1
270  IF I < L THEN 180         increment and branch back

280  IF F = 0 THEN 310         if in order, stop

290  L = L-1
300  GO TO 160
310  MAT PRINT P;
340  END
```

The last example underlines the fact that the more efficient and more elegant program is not always the simplest when written out.

ACHTUNG!

Alles Lookenspeepers

DAS COMPUTENMACHINE IS NICHT FUR GEFINGERPOKEN UND MITTENGRABBEN. IST EASY SCHNAPPEN DER SPRINGENWERK, BLOWENFUSEN, UND POPPENCORKEN MIT SPITZENSPARKEN.

IST NICHT FUR GEWERKEN BY DAS DUMMKOPFEN. DAS RUBBERNECKEN SIGHTSEEREN KEEPEN HANDS IN DAS POCKETS—RELAXEN UND WATCH DAS BLINKENLIGHTS.

A Crooked Shuffle

A Case Study in Debugging the Programmer

Alan Filipski

In an article on shuffling in the Jan.-Feb. 1977 issue of Creative Computing, John Jaworski considered the problem of generating the numbers from 1 to N in a random order without repetition (a "random permutation"). Both solutions given in that article have execution times on the order of N^2, *i.e.* shuffling 10N items would take about 100 times as long as shuffling N items for large N. My first reaction was that there is an obvious way to shuffle in linear time (time proportional to N for large N). It turns out that there is indeed such a way, but we have to be a little careful about what is "obvious." The following account traces the development of such an algorithm, pointing out some tempting fallacies along the way.

The germ of the idea is this: We first create an array containing the numbers from 1 to N in order. We then proceed to destroy that order by interchanging the contents of each location in turn with the contents of a location selected in some random fashion. To make this idea more precise, we could say

1. Generate an array A containing the numbers from 1 to N in order.

2. For each i from 1 to N:
Pick a random integer j between 1 and N and switch A_i with A_j.

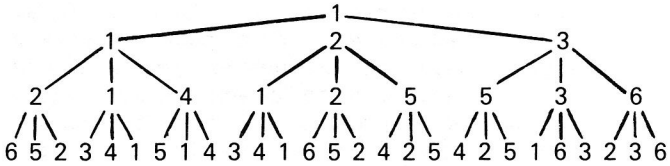
Thus every item gets switched at least once and on the average twice. This would be easy to program and takes linear time to execute. The method obviously mixes things up so thoroughly that we certainly must be getting random permutations. Of course, we could prove it if we wanted to, but proofs are just pedantic exercises, and besides, we have programming to do, right? Well, just for laughs, let's try to prove that this algorithm does what we want.

First, we should clarify exactly what we mean by the phrase "generating the numbers from 1 to N in a random order without repetition." The "without repetition" criterion is easy to verify because it is a property which must apply to *each* sequence generated. The "random order" criterion requires a little more thought, since it is a notion which applies to the entire class of permutations generated, but not to any single permutation (at least not without arousing some statistical and philosophical demons who are better left undisturbed). As a definition of "random order" we might venture to say that the probability of the number I appearing in the J^{th} position should be 1/N for all I and J between 1 and N. This insures that any number has an equal chance of appearing anywhere, so the program which satisfies this criterion must be generating all permutations at random, right? Wrong. Suppose N = 3. Then the possible permutations are:

$P_1 = (1\ 2\ 3)$ $P_2 = (2\ 1\ 3)$ $P_3 = (3\ 2\ 1)$
$P_4 = (1\ 3\ 2)$ $P_5 = (2\ 3\ 1)$ $P_6 = (3\ 1\ 2)$

Consider a program which outputs P_1 or P_5 or P_6, each with probability 1/3. This satisfies our proposed criterion, but is obviously not what we mean by a random shuffle, because the probability of generaging P_2, P_3, or P_4 is zero. This suggests that what we really want to say is that our program must generate any permutation with equal probability (probability 1/n! in fact, since there are N! different permutations.) Now that we know what we want, let's see how our program goes about producing it.

Consider the case when N = 3. The program starts with P_1. The first interchange transforms it to either P_1, P_2, or P_3 with probability 1/3 each. Two more interchanges are then performed on the result giving the final permutation. Since we have three choices at each of the stages, there is a total of 27 equally likely series of interchanges. Of course, some sequences of interchanges must produce the same result since only six different permutations are possible. We can represent these successive transformations by a tree as follows:



We note that at the final level, P_1, P_3, and P_6 occur four times each, while P_2, P_4, and P_5 occur five times each. The latter are therefore more likely to be generated than the former. Of course, if we were smart, we could have foreseen trouble just be observing that 6 does not divide 27 evenly.

So it appears that our algorithm does a rather slipshod shuffle. Well, what now? Is the idea bankrupt? Maybe noit. Consider

Department of Mathematics, Central Michigan University, Mt. Pleasant, MI 48859.

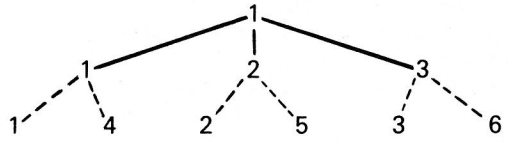
a modification of the technique: We start with the array A_1, A_2, A_3, . . . A_N containing the numbers from 1 through N in order. We begin as before by interchanging A_1 with the contents of a randomly selected location. We now want to set A_2 equal to one of the *remaining* items. This is the key to the rehabilitation of the algorithm. We accomplish this by selecting a random integer J between 2 and N and interchanging the contents of A_2 with A_J. Continuing in this way, our algorithm now becomes:

1. Generate an array A containing the numbers from 1 to N in order.

2. For each i from 1 to N: Pick a random integer j between i and N and switch A_i with A_j.

If we now display the situation for N = 3 in terms of our treee,

If we now display the situation for N = 3 in terms of our tree, we have:



which is exactly what we want, generating each permutation with probability 1/n!. We can now implement the algorithm with the following program:

```
100 DIM M(52)
110 LET N = 52
120 FOR I = 1 TO N
130 LET M(I) = I
140 NEXT I
150 FOR I = 1 TO N-1
160 LET J = INT(RND(0)*(N-I + 1)) + I
170 LET T = M(I)
180 LET M(I) = M(J)
190 LET M(J) = T
200 NEXT I
210 MAT PRINT M;
220 END
```

Thus we arrive at an efficient and simple solution to the original problem. As you may have guessed, however, the point of this paper is not the presentation of a shuffling algorithm which works in linear time (which can be found, for example, in Knuth's *Seminumerical Algorithms*) but rather an illustration of potential traps along the path of algorithm development. If you had (as I did) a tendency to swallow the argument that the first version of the algorithm "mixes things up so thoroughly that we must be getting random permutations," you have a bug in your quantitative intuition. This sort of bug is more insidious than any program bug since it potentially affects any algorithm you might develop. The existence of such bugs is not often publicized since it is ever the wont of mathematicians to display their creations in the austere beauty of their perfected form and to be ashamed of the false starts and jumpted conclusions along the way. (The exception here is the "paradox" which is such a dramatic and epidemic bug that it has entertainment value.)

If we are to make progress in exorcising these bugs, it behooves us to stop at least and recognize them for what they are. In the future, would we be more suspicious of a line like the "mix-em-up" argument? Is it clear that the picture of the tree leads to a proof in the case of the revised algorithm? Is it reasonable that N = 3 should yield a sufficiently general example to discredit the first algorithm, but that N = 2 should not? The consideration of such questions would be a first step in the debugging of the programmer.

Shuffling Revisited

The article on "Shuffling," in the Jan-Feb issue (page 77) drew a large response from readers who offered shorter or "more elegant" ways of solving the problem. Here are a few of the letters:

"More Elegant"

Dear Editor:

John Jaworski's article, "SHUFFLING", in the January-February issue contains a minor error in statement 180. The > symbol will produce a descending sort rather than the ascending sort shown in the before-and-after example. This has no real effect on the outcome except to reverse the order of the randomized integers.

Shown below are two routines which are more elegant than the shuffling technique (from the standpoint of requiring less iterations for a typical run and being more concise in code length):

The first uses a search technique borrowed from hashing algorithms rather than performing a sort.

```
100 DIM A(10), P(10)
110 FOR I = 1 TO 10
120 A(I) = I
130 NEXT I
```

A contains a table of integers, P will contain the integers in random sequence. The first loop puts the integers in A.

```
140 FOR I = 1 TO 10
150 J = INT (10*RND + 1)
160 IF A(J) > 0 THEN 210
170 J = J + 1
180 IF J < 11 THEN 160
190 J = 1
200 GO TO 160
```

Generate a random integer use this integer to access A Scan through A until you find an integer which has not been used yet.

```
210 P(I) = A(J)
220 A(J) = 0
230 NEXT I
```

Place the next integer in the output table and remove this integer from A

```
240 MAT PRINT P
250 END
```

Print P when all integers are moved.

The second routine shows how this same function appears in APL:

David D. Keefe
Tillson, NY

"Each Loop Used Only Once"

Dear Editor:

On reading "Shuffling" by Jaworski in Creative Programming Techniques, January-February 1977 issue, I notice a sort is required. For longer lists, this can be a time-consuming routine. Here is a routine to shuffle 52 cards in one pass. Cards are picked one at a time and each of the remaining cards has an equal chance of being picked.

```
100 DIM M(52)
110 N = 52
120 FOR I = 1 TO N
130 M(I) = I
140 NEXT I
```

Enter numbers 1 to N in list in order.

```
150 FOR I = 1 to N-1
160 R = (N + 1-I)*RND(1)
170 R = INT(R) + I
180 T = M(R)
190 M(R) = M(I)
200 M(I) = T
210 NEXT I
```

Pick number R between I and N.

Exchange entries I and R.

Each loop is used only once.

James Murphy
Associate Professor
California State College,
San Bernadino, CA 92407

"Simpler and Smaller"

Dear Editor:

The article by John Jaworski on "Shuffling" was very interesting. However, I am unimpressed by the little "moral" at the end. Several years ago I constructed a card-shuffling program based on an explanation of permutation theory based on a mail-clerk and pigeon holes. I don't remember the source of the explanation or its precise details, but I do remember the algorithm. Translated to BASIC it looks something like this:

```
DIM M(10)
FOR I = 1 TO 10          Initialize the array—this step
M(I) = I                 is only required once and the
NEXT I                   program can be used to
                         generate as many permuta-
                         tions as you wish.

FOR J = 1 to 9
K = M(J)
L = INT ((11-J)*RND + 1)
M(J) = M(L + J - 1)
M(L) = K
NEXT J
```

As you can see, the algorithm chooses each element of the permutation randomly from the numbers not previously chosen. The advantages over sorting are: (1) less memory is required (only one vector instead of 2), (2) fewer exchanges per permutation (no sorting program can beat N − 1 consistently), (3) no comparisons at all and (4) the program itself is much simpler and smaller.

The January February issue was my first experience of your magazine—I enjoyed it thoroughly! Keep on computing!

Dean Ritchie
Systems Programming Manager
Computing Center
Washington State University
Pullman, WA 99163

"Requires Less Memory and Time"

Dear Editor:

This letter could be headed "A Better Way to Shuffle." I was disappointed to see that John Jaworski omitted one easy shuffling technique—random indexing—from his treatment of BASIC programming, and wish to fill the void. To shuffle an array using random indexing is to choose elements by using random numbers to calculate addresses. The following BASIC statement will calculate the address of one of an N-element array with subscripts ranging from 1 to N. If your BASIC interpreter recognizes the zeroth element of an array, then the statement will have to be changed to avoid wasting an array element.

```
I = INT(N*RND(0) + 1)
```

After the Ith element is removed from the array and stored in a safe location, the array is packed by moving the top elements down one space, and N is decremented by 1. Another element is selected using the same method, and the process repeated until the array is used up. You might think two large arrays would be needed, one to hold the source array of elements, and one to hold the shuffled array, but that isn't so. Remember that after the Ith element was selected, the remaining elements were packed together to eliminate the gap. That left a gap at the top of the array where the element would fit nicely. Packing the array isn't difficult, either. Because the shuffled array is supposed to be in random sequence, it really doesn't matter what order the source array is in. To pack the array, remove the unselected upper element from the top of the array and plug the gap. Putting it all together for a program to print nine digit numbers, with no two digits the same, yields the following BASIC code:

```
 100 DIM A(9)
 200 REM FILL THE ARRAY WITH
 300 REM THE DIGITS FROM 1
 400 REM TO 9
 500 FOR I = 1 TO 9
 600 LET A(I) = I
 700 NEXT I
 800 REM THE SHUFFLING ROUTINE
 900 FOR I = 9 TO 2 STEP -1
1000 LET J = INT(I*RND(0) + 1)
1100 IF J>I THEN 1000
1200 LET T = A(J)
1300 LET A(J) = A(I)
1400 LET A(I) = T
1500 NEXT I
1600 FOR I = 1 TO 9
1700 PRINT A(I)
1800 NEXT I
1900 END
```

This program requires less memory and time than the routines provided by Mr. Jaworski. Speed and space-saving are important, especially in a program like BLACKJACK which shuffles a 52-card deck several times.

William R. Hamblen
946 Evans Rd.
Nashville, TN 37204

"At Random"

Dear Editor:

While looking through the January/February *Creative Computing*, I noticed the "Shuffling" article (J. Jaworski, p.77), thought, "There, but for the grace of Iverson, goes 10?10," and turned the page. But then, upon a closer reading of the magazine, I discovered the same technique advocated on the very facing page! And with the same ineluctable bubble sort! This was too much. Even with a good sort, the program is inefficient. The obvious way to shuffle 10 or any number of n numbers is: a) pick one at random b) pick one of those remaining c) continue until none are left. Since the two sets, picked and unpicked, will always total 10 (or however many) numbers, we just move the boundary through the array, exchanging the number whose place we want with the one we wish to put there. BASICly:

```
100 DIM A(10)                          A is 1, 2, . . . ,10.
110 FOR I = 1 TO 10
120 A(I) = I
130 NEXT I
140 FOR I = 1 TO 9                     I is the boundary.
150 K = I + INT (RND*[11-I])           K is a random number
160 T = A(I)                           from I to 10.
170 A(I) = A(K)                        Exchange
180 A(K) = T
190 NEXT I
200 MAT PRINT A;                       Done.
210 END
```

Using the sorting method squares the time (depending on the sort) and doubles the space (code and arrays) that the program requires.

J. Storrs Hall
New Brunswick, NJ

Shell-Metzner Sort vs Hart Sort

Dear Editor:

As sorting is an operation frequently used on our computer (8K,PDP11/10) I was interested in the article by Richard Hart (Jan-Feb 1978, pp. 96-101). At present we are using the Shell-Metzner sort which was described by John P. Grillo (Nov-Dec 1976, pp. 76-80).

This latter sort requiring only about a quarter of the statements required by method described by Hart. On comparing sorts within the limitations of our BASIC—maximum array size 255—I find the Shell-Metzner sort is also faster in sorting randomly generated numbers.

Number sorted	*Shell-Metzner (Time in seconds)*	*Method described by Hart*
50	21	27
100	48	61
200	121	135

Within these values the Shell-Metzner wins on two counts (1) compactness of the algorithm and (2) faster sorting.

Perhaps someone who has a computer that can handle larger arrays could look at comparisons above 200.

Pat Fitzgerald
Winchmore Irrigation Research Station
Ministry of Agriculture & Fisheries
Ashburton, New Zealand

Programming Techniques: File Structures

John Lees

The primary use of general-purpose computers is data processing, and most of data processing is file handling. To make effective use of your computer, sooner or later you'll have to deal with files. So here's an introduction to the wide world of file structures.

To begin with, a few definitions: A *file* is an organized collection of related information. (A collection of files can be called a data base.) A file consists of *records,* all of which usually have the same basic structure. Each record can, in turn, be subdivided into *fields,* or *elements.*

A file can exist in memory, on paper tape, cassette, magnetic tape, disk, or any other type of storage. Storage can be considered in two categories, that allowing only sequential access (all kinds of tape and some primitive disk systems); and that allowing random access (semiconductor or core memory and most disk systems). Some tape systems purport to have random access, but random access on tape can only be achieved with considerable access time and space overhead.

Sequential File

The simplest and most common file structure is *physical sequential.* Records are arranged in some order, one after another in the file, and are physically accessed in that order. Tape files are by their very nature physically sequential.

Let's consider an example of a sequential file. Say you want to keep a catalog of all the books in your library, or all your record albums, musical scores, paintings, or any similar item. A record in such a file might look like this:

AUTHOR	TITLE	PUBLISHER	ADDRESS	BINDING	PAGES	PRICE	DATE

The file would consist of one such record for each book in your library. Probably the order in which you would choose to keep such a file stored would be alphabetical by the author field (which would be the *key field* for the file), but of course you could store it in any order you wish. The order you decide to use is important though, since you don't want to have to sort the file each time you use it.

Once you set up your library file, you'll want programs to add books or delete books and possibly to allow you to modify a record, although you could get by with deleting, then adding, to modify. You'll want the capability to print, say, all titles by one author, or by one publisher, or all hardbound books, or all books published in 1974.

Two Transports

If you're using tape, you'll need two transports to be able to keep the file in order, when you add or delete records. To do that, you'd write a program to take all the additions or deletions, sort them and keep them in memory. Then your program would read from one tape, writing records out to the other tape until it reaches the point for an add or delete. It would do the add or delete and then keep on reading/writing until the entire file had been processed.

You may have noticed that a lot of space is being wasted in our sample file. There is only a small number of different publishers, yet that information is repeated in every record. Very wasteful! To save space, we could create two files, one file consisting of a modification of the records we already have, with the publisher information replaced with a code:

AUTHOR	TITLE	P-CODE	BINDING	PAGES	PRICE	DATE

and another file containing the publisher information:

P-CODE	PUBLISHER	ADDRESS

These two files could be used in this way. Put the Publisher file first on the tape, followed by the Library file. As the first step in using the Library file, read the Publisher file into the memory since it is relatively small. Now, as records are read from the Library file, the publisher code can be matched up with one in the Publisher file and the information in the record in the Publisher file used to print the book listing. This look-up will be fast since it is done in memory. The small amount of extra processing time used is well justified by the savings in file space. This same principle can be applied whenever a field contains often-repeated information. Of course the information must be longer than the code used to replace it. It wouldn't pay to do this with the date field, for instance.

Tight Space

If you're real hard-up for space, the leading "1" need not actually be stored in the date field. Similarly, don't store a "$" or even the decimal point in the price field. Have the program add them when it adds the publisher information to the book listing being printed. You can save even more bytes by storing the price and date in binary and converting. But don't get carried away if you don't need to save the space.

All well and good, but what if you have collections, such as science-fiction anthologies, and want to be able

to find authors and titles of stories in the collections? You could do about the same thing we did with publisher information. Add a code to the author field:

AUTHOR	B-CODE	TITLE	P-CODE	BINDING	PAGES	PRICE	DATE

This new code could mean if 0 then the book is not a collection, else the code would match up with a set of records in a short-story file which would give the contents of that collection.

But that isn't really what you want. That scheme will let you list the contents of a collection, but is of no help in finding out if you have in your library a short story that only appears in a collection. What to do? You could, instead of having a separate short-story file, include these records in the main file. Avoiding redundant information, you would now have a file consisting of three different types of records:

TYPE 0	AUTHOR	TITLE	P-CODE	BINDING	PAGES	PRICE	DATE

TYPE 1	B-CODE	AUTHOR	TITLE	P-CODE	BINDING

TYPE 2	B-CODE	AUTHOR	TITLE

A type 0 record would be a normal book. A type 1 record would be a collection and such a record would contain an additional field with a book code, which would match a book code in type 2 records containing the authors and titles of the short stories in the collection. To make this file easily usable with a sequential storage medium, you'd probably want to group all the records together by types (almost, in effect, giving three files), in alphabetical order by author within type.

Now if you want to see if K is in your library, the program would look for a record of any type with K in the author field. If the record(s) found with K are of type 2, then the program would also look for a type 1 record with matching b-code and tell you what collection the story by K appears in. If you think that this kind of thing could take a very long time on a cassette, you're perfectly correct. But what's your hurry?

Faster, Faster

Well, maybe you're writing this system for the school library and you have a legitimate reason for wanting the search for a book to take less than half an hour. Hopefully you can get a disk or two, because you've exceeded the capabilities of a sequential-file structure. The rest of the structures we're going to discuss require random-access devices.

The drawbacks to the plain old sequential file are obvious. If you're on Heinlein, you know that Vonnegut is somewhere further on and that you've passed Ellison. Vonnegut you'll eventually come to, but Ellison can only be reached by going back to the beginning and starting over again. An unhandy state of affairs if speed is of any importance at all.

Index

So, enter the next bright idea in file structures, the index. Although the combination of indexes and sequential files will not, as IBM once tried to convince the world, solve all problems, it does help a little. Imagine a dictionary with no way of telling where each letter begins and you'll quickly appreciate the utility of an index. The idea is the same with a file. You have the master file and an *index file* which contains the information on where certain categories begin in the master file. This could be in terms of record number, memory address, disk sector, or (shudder) tape block. Now if you want to find Heinlein, the program looks in the index file and goes right to the beginning of the H's. This is of limited use on tape, since you still have to move all that tape past the read head slowly enough to count blocks.

With our example, you could also have an index to help find the groups of short-story records and even the records for the collections themselves. So your program could go right to the record or group of records. A diagram for such a file system might look like this:

Alphabetical Index

A	LOCATION

•
•
•

Z	LOCATION

Short Story / Collections Index

B-CODE	LOCATIONS OF SHORT STORY RECORDS	LOCATION OF COLLECTION RECORD

•
•
•

B-CODE	LOCATIONS OF SHORT STORY RECORDS	LOCATION OF COLLECTION RECORD

Library Master File

TYPE 0	AUTHOR	TITLE	P-CODE	BINDING	PAGES	PRICE	DATE

TYPE 1	B-CODE	AUTHOR	TITLE	P-CODE	BINDING	PAGES	PRICE	DATE

TYPE 2	B-CODE	AUTHOR	TITLE

Publisher File

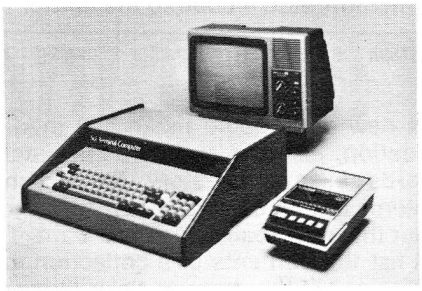
P-CODE	PUBLISHER	ADDRESS

Once you've figured out that conglomeration, you'll see that it saves a lot of work, at the expense of a little storage space. Using the indexes may speed things up, but a lot of sequential processing is still required, and the records within groups must still be kept in alphabetical order, thus requiring a lot of insert overhead. (Deletes are simple. Just adopt the convention that a type 3 record isn't there and so mark "deleted" records, every once in a while collecting the garbage and squishing things together.) Also, there are a couple of little bugs in that file system and a very high maintenance cost associated with updating all those index records if any of the Master file records are moved, as they will be each time an insert is performed.

It is possible to get away entirely from any reliance on sequential ordering, at the expense of a little more storage and a little more processing time. But processing time is cheap and maintaining a sequential file is a nightmare when you don't have much memory. So let's move on into the realm of list structures, linked lists, rings, trees, hierarchical files and such things. You ain't seen nothing yet! ■

Apple II $1195

Heathkit H-8 $375



SOL-20 $1600

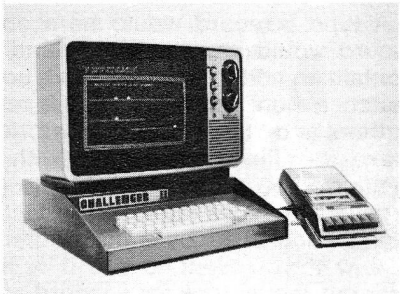
Which of these computers is for you?

Exidy Sorcerer $895

Bally Arcade $399

CREATIVE COMPUTING magazine is Number 1 in hardware, software and system evaluations. In-depth, thorough evaluations give you the facts **before** you buy. **Creative Computing** was the first to review these now popular systems: Radio Shack TRS-80, Exidy Sorcerer, VideoBrain, Heath H-8, Bally Basic, OSI Challenger, and many others. More important, we also review peripherals and software from independents as well as manufacturers.

Radio Shack TRS-80 $599



Ohio Scientific C2-4P $598

And what are you going to do with it?

CREATIVE COMPUTING has long been Number 1 in applications and software for micros, minis, and time-sharing systems for homes, schools and small businesses. Loads of applications every issue: text editing, graphics, communications, artificial intelligence, simulations, data base and file systems, music synthesis, analog control. Complete programs with sample runs. Programming techniques: sort algorithms, file structures, shuffling, etc. Coverage of electronic and video games and other related consumer electronics products, too.

Just getting started? Then turn to our technology tutorials, learning activities, short programs, and problem solving pages. No-nonsense book reviews, too. Even some fiction and foolishness.

Subscriptions: 1 year $15, 3 years $40. Foreign, add $9/year surface postage, $26/year air. Order and payment to: Creative Computing, attn: Gretchen, P.O. Box 789-M, Morristown, NJ 07960. Visa or Master Charge acceptable by mail or phone; call **800-631-8112** 9 am to 5 pm EST (in NJ call 201-540-0445).

CREATIVE COMPUTING also publishes books, games, art prints, and T-shirts for the computer enthusiast. The most popular book of computer games in the world, **Basic Computer Games** is a Creative Computing book — only $8.50 postpaid.

And now, Creative Computing also produces and markets software for personal computers on tape cassette and floppy disk.

If your dealer does not carry the full line of Creative Computing products, write "catalog" on your order and we'd be happy to send you one free.

creative computing

P. O. Box 789-M, Morristown, NJ 07960