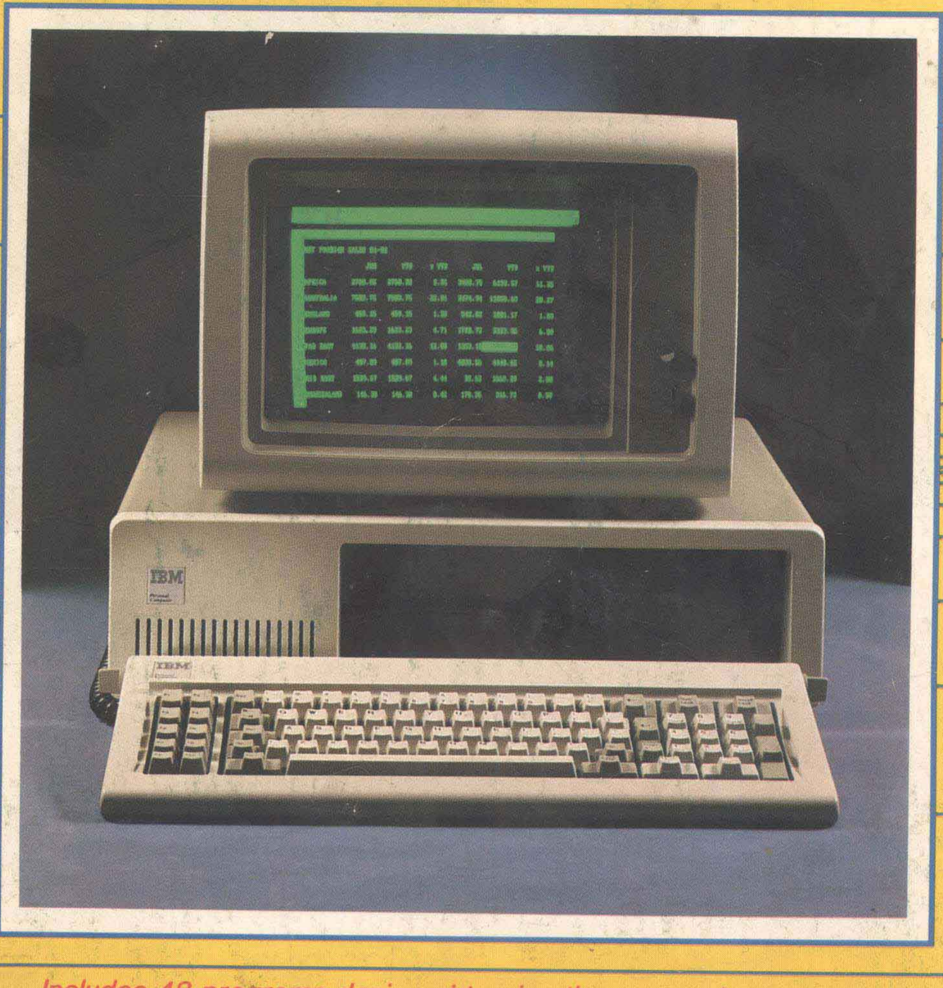
MICROPOWER SERIES

$16.95

DATA AND FILE MANAGEMENT for the IBM Personal Computer

John P. Grillo / J. D. Robertson



Includes 48 programs designed to give the more advanced user techniques for managing stored data.

MICROPOWER SERIES

DATA AND FILE MANAGEMENT for the IBM Personal Computer

John P. Grillo / J. D. Robertson
Bentley College
Waltham, Massachusetts

wcb
Wm. C. Brown Company Publishers
Dubuque, Iowa

Consulting Editor:
Edouard J. Desautels
University of Wisconsin-Madison

Library of Congress Catalog Card Number: 83–70098

ISBN 0–697–09987–3

2-09987-03

Third Printing, 1983

Printed in the United States of America

DATA AND FILE MANAGEMENT for the IBM Personal Computer

To Donald and Nicklaus

Introduction

What is a data structure? How can the understanding of a search technique help you to write a better genealogy program? What in the world are stacks, queues, deques, and trees? Do you lose the contents of a file if you invert it?

There is a reason why programmers ask these questions. The questions stem from a lack of understanding of some fundamental programming concepts that deal with data. Most programmers rely on a background of vendor manuals and perhaps one or two formal courses in BASIC. They feel confident in their ability to deal with lists, arrays, subscripts, and some sequential searches. But unfortunately this level of programming expertise, this repertoire of techniques, is not enough to be helpful in writing good, efficient software.

In the classroom and in the world of consulting, we are quick to point out the crucial importance of writing usable programs, that is, programs that benefit the user. We also emphasize that the programmer is rarely the only user of a good program, because a good program is by definition one that is used by many people. A good programmer must take pains to write programs that are easy to use. A good program has the following properties:

- It should be structured well, so that its author and all other readers of the program feel confident in being able to change parts of it — add, delete, or modify modules — without adversely affecting the untouched portions.
- It should be documented well, so that its logic can be understood easily.
- It should be written to be interactive whenever this process can benefit the user.
- It should use files if these media for storing information are appropriate.
- It should make efficient use of the computer through the proper use of algorithms that minimize sorting and searching times, minimize disk accesses, avoid excessively large memory arrays, and avoid inappropriate data storage techniques.

The first three of these properties are discussed thoroughly in the book, *Techniques of BASIC for the IBM Personal Computer*. The last two properties are the focus of this book.

In most colleges and universities there exists a course called Data Structures. It is taught after two semesters of programming and its intent is to refine the students' technique and introduce the commonly used procedures that have been developed over the years to make a program run more efficiently. The course covers pretty much what this book covers, and in pretty much the same order. After mastering these techniques, these same students seem to be able to deal with masses of data, either in memory or on files, with considerable ease and success. The Data Structures course, more than any other course in their formal training, prepares them to writer user-oriented programs with direct applicability in their future industrial exposure.

The popularity of the IBM Personal Computer is increasing continually. As a result, the industry pundits predict it to become one of the three or four top home computers of the mid-1980s. It is likely that most individuals will purchase a small-scale IBM PC system, even though their eventual configurations could include a wide range of peripherals. The system we used for the programs included in this book is such a modest system. It has 64K bytes of memory available to the user, two 320K double density disk drives, and a monochrome display unit. We also had available a Brother HR-1 letter quality printer, which we used to produce all included listings. We used Version 1.10 of the IBM Disk Operating system (DOS) and its accompanying Advanced BASIC. We have felt unrestricted in our applications programming using this configuration.

One of the advantages we have discovered in our diverse dealings with microcomputer literature is the frequently available software that accompanies books and magazines in the form of cassettes and diskettes. Our publisher, Wm. C. Brown, has made available to our readers on diskette all of the programs described and listed in this book. You should strongly consider the purchase of the software on diskette, if for no other reason than to save you the tedium of keying in the programs. Of course, that latter procedure is frought with the hazards of error generation through typographic slips.

In this book we have made a sincere effort to simplify and demystify the subject of data structures. We call the book *Data and File Management* because we are trying to make a point: The topic of data structures is useful only as long as it relates to the practical aspects of how to manage data. The programs we include as examples should serve to show you how these techniques can be useful in many common applications. We sincerely hope that some of these intrigue you enough that you will adapt them in some novel fashion and that you have as much fun using them as we had writing them.

Contents

1 Pointers

BASIC has become the popular problem-solving language for microcomputers for a wide variety of reasons, not the least of which is its inherent simplicity. It is easy to learn and to use. Consider the long list of high schools and colleges that teach BASIC as a way to introduce the computer to novices.

One important reason for BASIC's popularity is often overlooked: The language is highly flexible. By this we mean that BASIC will allow programming constructs that are difficult or impossible to manage in another language. A case in point is an array's subscript. In BASIC, the only rule is that the subscript be a numeric expression, while in many versions of FORTRAN, for example, the subscript form is limited to but a few very simple variations.

Subscripts

The subscript is the programmer's pointer into an array, and as such must be capable of as much variation as possible.

The subscript as a pointer in its most elemental form is simply a way to access a given array element. For example, consider this segment of code:

```
100 DIM D(50)
110 P=17
120 V=4
130 D(P)=V
```

The variable P assumes the role of pointer to the array D in line 130 when it is used as a subscript. The overall effect is to store the value 4 into the 17th location of D.

Suppose this line were added:

```
140 D(D(P))=237.8
```

This time, the pointer to the array D is the variable D(P), which in itself uses a pointer. P=17, as defined in line 110 above. D(P)=D(17)=4, from lines 120 and 130. D(D(P))=D(4), so define D(4) as 237.8. Thus the value 237.8 is stored in D(4). This form of addressing an array is called *indirect addressing*, because the computer determines the final destination by proceeding through an intermediate location that points to the value to be transferred.

In order to complete this introduction to pointers, we should point out that most versions of FORTRAN don't allow subscripted variables to have subscripted variables as subscripts. Try saying that ten times fast!

The programs which we have selected to include in this chapter to exemplify the use of pointers all have a common concern: Where do the array pointers come from? You will discover that array pointers can be selected from a pool, generated at random, or calculated. This differs from the more common source of array pointers, such as a FOR-NEXT loop index or a counter.

Nim, G1A, and Heuristic Programming

The first program, G1A, is an interesting variant of the game of Nim. The two Nim players remove from one to three objects from a starting set of 13 objects. The player who removes the last object loses. The game can always be won by the second player if that player remembers one rule: Always leave a pile with the number of objects left equal to 9, 5, or 1.

In the early days of computers, much discussion centered on how to program these machines to assume the characteristics that would make them seem intelligent. The area of interest, called *Artificial Intelligence*, or AI, was born. One of the techniques for simulating intelligence was given the name *heuristic programming*, or programming with the intent to discover or reveal an underlying principle.

In 1965, H. D. Block in an article in *American Scientist* described a machine that would "learn" to play the game of Nim with a winning strategy. The machine, originally called G1, did a pretty fair job of imitating the way we humans learn. At first, it would play in seemingly random fashion. After many games it would give up the current game as lost before the game was over. It was as if it had discovered the futility of continuing that game. Then, many games later, it became unbeatable. It had "learned" the way to win.

To speed up the learning process, Block altered G1 and created G1A. We present this machine to you now as a program. The program sets up all possible moves for itself in four cups. Think of each cup containing three slips of paper, marked 1, 2, and 3. As a game proceeds, G1A "draws" its move from the appropriate cup at random.

When the flow of the game determines that G1A has lost, the last "draw" that G1A made before losing is marked with a −1. From that point on in the series of games, this move will not be made. Eventually, all cups' moves contain a −1 except those that lead to wins, and those will be the only plays G1A makes.

Part of the fun of this program is to trace G1A's progressively better play. We leave this as an exercise for the reader.

Some features of the program are worthy of special mention.

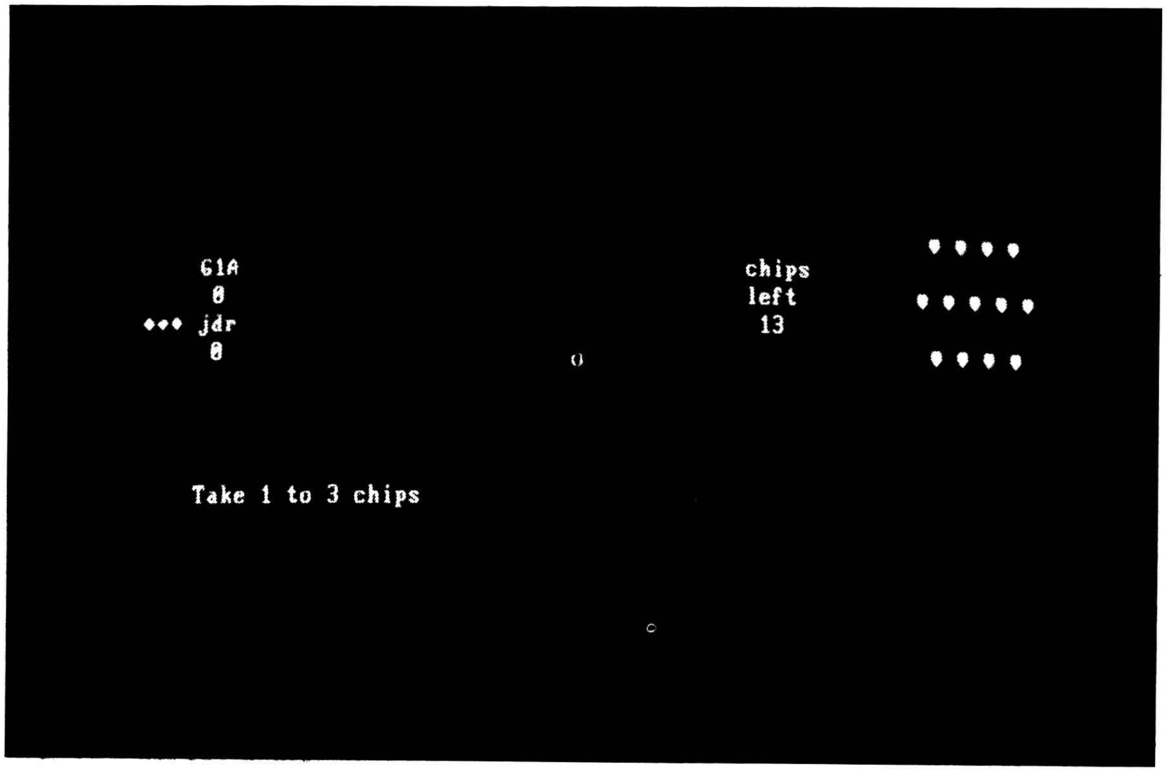
- All user inputs are programmed with the INKEY$ function. Study the portion of the program that asks for the user's initials. This section uses INKEY$ to build an input string three characters long without the use of the ENTER key.
- The LOCATE instruction is used extensively to display the game's status as it changes.
- Graphic characters are used to display the chips. Note that a special effort is made to remove the chips randomly from the pile.
- The messages that G1A displays give it a personality. If G1A loses, it responds in modest lower case. If G1A wins, it responds in an obnoxious and pretentious upper case.

```
10 'filename: "g1a"
20 ' purpose: To play the game of "NIM" heuristically
30 '  author: jdr & jpg 9/82
40 DEFINT A-Z
50 DIM CUP(4,3), CY(13), CX(13)
60 DIM PERM(6,3)                '6 permutations of 3 digits.
70 '                            His win or Our win messages.
80 DIM H1$(6), H2$(6), O1$(13), O2$(13), O3$(13)
90 RANDOMIZE: CLS
100 '                                 Flash the title screen.
110 LOCATE 5,1: PRINT STRING$(80,1) 'smile
120 LOCATE 7,25: PRINT "N I M   W I T H   G 1 A"
130 LOCATE 9,1: PRINT STRING$(80,1)
140 '              Fill each set of 3 cups with digits 1,2,3.
150 FOR I=1 TO 4
160   FOR J=1 TO 3: CUP(I,J)=J: NEXT J
170 NEXT I
```

```
180 '  Fill the PERM array with all permutations of 1,2,3.
190 FOR I=1 TO 6
200   FOR J=1 TO 3: READ PERM(I,J): NEXT J
210 NEXT I
220 '        Fill His wins messages with gracious pap.
230 FOR I=1 TO 6: READ H1$(I), H2$(I): NEXT I
240 '      Fill Our wins messages with scathing scorn.
250 FOR I=1 TO 13: READ O1$(I), O2$(I), O3$(I): NEXT I
260 LOCATE 12,1: PRINT STRING$(54,32): WHO$=" / "
270 '**                         Set up the player's name.
280 LOCATE 12,3: PRINT "enter your initials";: T$=INKEY$
290   IF T$="" THEN 270
300   WHO$=WHO$+T$: LOCATE 12,25: PRINT WHO$
310     IF LEN(WHO$)<>6 THEN 270
320 GOSUB 5000: WHO$=RIGHT$(WHO$,3): CLS
330 '**                                 Set up the screen display.
340 LOCATE 7,10: PRINT "G1A";    '    The computer's name is "G1A".
350 LOCATE 9,10: PRINT WHO$; '     Print the player's name.
360 LOCATE 8,10: PRINT WIN;    '   Print the number of player's losses.
370 LOCATE 10,10: PRINT LOSE;      ' Print the number of player's wins.
380 GOSUB 1000                   '  Blank out previous chips.
390 '**                                 Position the chips.
400 LOCATE 7,6: PRINT "   ";    '    Reverse marker for player turn.
410 LOCATE 9,6: PRINT STRING$(3,4);    '     Mark next player.
420 '**                        Get player's response.
430 LOCATE 15,10:PRINT "Take 1 to 3 chips";: T$=INKEY$
440   IF T$="" THEN 420
              ELSE TAKE=VAL(T$): LOCATE 15,30: PRINT "/ ";T$: GOSUB 5000
450   IF TAKE<1 OR TAKE>3 OR TAKE>CHIPS
         THEN GOSUB 6000: LOCATE 15,10: PRINT "improper move";: GOSUB 5000:
              GOTO 420
460 GOSUB 2000
470   IF CHIPS<=0 THEN GOSUB 4000: GOTO 330
480 '  G1A's move
490 GOSUB 6000: LOCATE 9,6: PRINT "   ";
500 LOCATE 7,6: PRINT STRING$(3,6);
510 '          A is the random permutation selector.
520 GOSUB 5000: A=INT(RND*6+1)
530 FOR I=1 TO 3
540 '               Check the Ath. cup.
550 '      B is the remainder:  CHIPS modulo 4.
560 K=PERM(A,I): B=CHIPS-4*INT(CHIPS/4)
570   IF B=0 THEN B=4
580 '    If this cup is not empty, check the others.
590 '      If it is empty, check others.
600   IF CUP(B,K)<=0 THEN NEXT I: CUP(BCUP,KCARD)=-1: HOW=1:
                         GOSUB 3000: GOTO 330
610 TAKE=CUP(B,K)
```

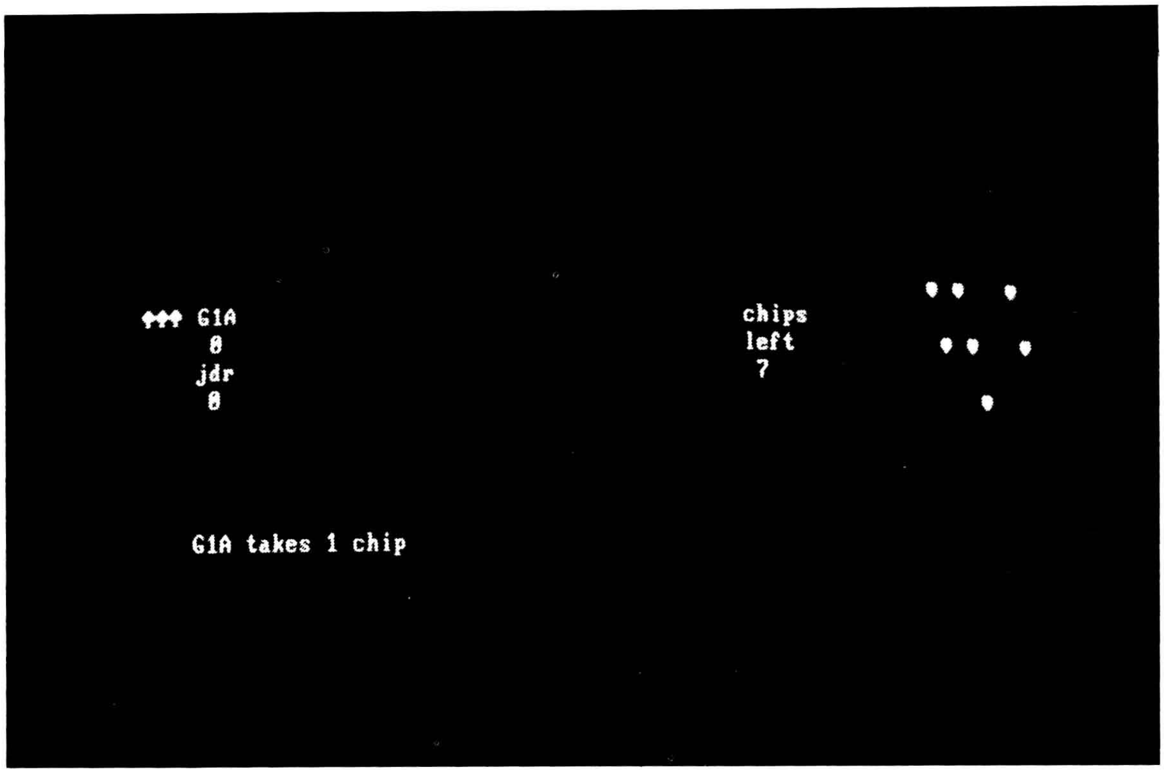
```
620 '     If this cup is not empty, check others.
630   IF TAKE>=CHIPS THEN CUP(B,K)=-1: HOW=2: GOSUB 3000: GOTO 330
640 ' Note proper grammatical display. 'chip' or 'chips'.
650 LOCATE 15,10: PRINT "G1A takes"; TAKE; LEFT$("chips",4+INT(TAKE/2)),
660 GOSUB 5000: GOSUB 5000
670 BCUP=B: KCARD=K
680 GOSUB 2000: GOTO 390
1000 '****              Subroutine to set up chips.
1010 N=0
1020 FOR I=-1 TO 1
1030   L=ABS(I): M=62+L
1040   FOR J=1 TO 5-L
1050     N=N+1: CX(N)=M+J+J: CY(N)=I+I+8
1060     LOCATE CY(N),CX(N): PRINT CHR$(3);
1070   NEXT J
1080 NEXT I
1090 CHIPS=13: LOCATE 7,51: PRINT "chips";: LOCATE 8,51: PRINT "left";
1100 LOCATE 9,51: PRINT CHIPS;: R=INT(RND*13+1): P=1+INT(RND*11+1)
1110 RETURN
2000 '****          Subroutine to remove 1 to 3 chips.
2010 FOR I=1 TO TAKE
2020   R=R+P
2030     IF R>13 THEN R=R-13
2040   LOCATE CY(R),CX(R): PRINT " ";
2050 NEXT I
2060 CHIPS=CHIPS-TAKE: LOCATE 9,51: PRINT CHIPS;
2070 RETURN
3000 '****       Subroutine to print loss message.
3010 GOSUB 5000: GOSUB 5000: CLS
3020 A=INT(RND*6+1): LOCATE 7,1: PRINT STRING$(80,254)
3030   IF HOW=2 THEN LOCATE 9,25: PRINT "G1A "; H2$(A);" acknowledges defeat"
3040   IF HOW=1 THEN LOCATE 9,25: PRINT "G1A "; H1$(A); " concedes the game"
3050 LOCATE 12,1: PRINT STRING$(80,254): LOSE=LOSE+1
3060 GOSUB 5000: GOSUB 5000: GOSUB 5000: CLS
3070 RETURN
4000 '****      Subroutine to print win by G1A.
4010 CLS: A=INT(RND*13+1): B=INT(RND*13+1): C=INT(RND*13+1)
4020 LOCATE 7,1: PRINT STRING$(80,254)
4030 LOCATE 9,15: PRINT " THE "; O1$(A); " G1A HAS ";
4040 PRINT O2$(B);" THE ";O3$(C);" ";WHO$
4050 LOCATE 11,1: PRINT STRING$(80,254)
4060 WIN=WIN+1
4070 GOSUB 5000: GOSUB 5000: GOSUB 5000: CLS
4080 RETURN
5000 '*****     Subroutine to mark time.
5010 FOR I=1 TO 1500: NEXT I
5020 RETURN
```

```
6000 '****    Subroutine for blanking out.
6010 LOCATE 15,1: PRINT STRING$(80,32);
6020 RETURN
7000 '****       d a t a   s t a t e m e n t s
7010 '
7020 '
7030 DATA 1,2,3,1,3,2,2,1,3,2,3,1,3,1,2,3,2,1
7040 '      Data for G1A losses
7050 DATA cordially, respectfully, graciously, politely
7060 DATA affably, humbly, congenially, modestly
7070 DATA meekly, amicably, courteously, agreeably
7080 '      Data for G1A wins
7090 DATA AWESOME, ANNIHILATED, PROSAIC, DREADED
7100 DATA EXTERMINATED, VAPID, PUISSANT, OBLITERATED, SLUGGISH
7110 DATA EMINENT, DEMOLISHED, DOLTISH, EXALTED, CONQUERED, OBTUSE
7120 DATA INTREPID, VANQUISHED, INFERIOR, SPLENDID, DEVASTATED
7130 DATA INSIPID, SAPIENT, EXTIRPATED, MAWKISH, ERUDITE, SUBJUGATED
7140 DATA BUNGLING, FORMIDABLE, CRUSHED, FLACCID, REDOUBTABLE
7150 DATA FLATTENED, INEPT, BRILLIANT, STOMPED, IGNORANT
7160 DATA MAGNIFICENT, DESTROYED, STUPID
9999 END
```



```
      G1A                                        chips     ♦ ♦ ♦ ♦
       0                                         left
  ♦♦♦ jdr                                         13      ♦ ♦ ♦ ♦ ♦
       0
                                                           ♦ ♦ ♦ ♦

      Take 1 to 3 chips
```



```
♠♠♠ G1A                                    chips
     0                                     left
    jdr                                     7
     0



     G1A takes 1 chip
```

Monte Carlo Methods

The next program, JOBSTEPS, demonstrates the use of random numbers as pointers to distribute an array's contents. The function of the program is to determine two sequences of hypothetical machining operations. Each of the sequences is to be assigned to one of two workers, with a sense of fairness requiring that the total time for the machining operations each is assigned be as closely matched as possible. There are as many as 25 various operations the two workers are qualified to do, and each worker can be assigned any number of these operations, as long as they both work the same total amount of time. Their boss, the user of this program, inputs the amount of time allotted, for example four hours. The program prints out two work schedules, each containing machining operations, or tasks. No task on one list appears on the other.

The technique of successively scrambling an array's contents, then checking to see if this order produces a better solution than a previous one, is an example of the *Monte Carlo Technique*, named after the famous casino at Monaco.

The solution of this problem is not trivial. To come up with such a schedule with paper and pencil takes the better portion of an hour. What the computer does in the program JOBSTEPS is to select at random a set from 25 possible operations and sum their times. As soon as the set exceeds the total time the boss dictates, for example four hours, the total time is displayed. The boss can elect to have the computer program select another set closer to the four hours, or accept that one.

When the first worker's schedule has been determined in this manner, the program uses that amount of time as a target and randomly selects from the unused tasks another schedule for the second worker. The computer displays its first try, and the boss can accept or reject it. A rejection forces the computer to come up with a better schedule. Each try that is closer to the target (worker 1's schedule) is displayed for the boss to accept or reject. When the boss finally accepts that run, both workers' schedules are displayed, with each total time shown.

```
10 'filename: "jobsteps"
20 ' purpose: Monte Carlo selection of job operations
30 '  author: jpg & jdr 9/82
40 '
50 DIM T$(25), T(25), K(25)
60 'T$ = operation              T = time per operation, min.
70 'K  = selected pointer
80 RANDOMIZE : CLS
90 '                                      Read tasks, times.
100 INPUT "What is target time (in minutes)";TT
110 FOR I=1 TO 25
120   READ T$(I), T(I): K(I)=0
130 NEXT I
140 S=0
150 T1=10   'Set difference between returned time S and TT.
160 GOSUB 1000
170 PRINT "Suggested time is"; S
180 LINE INPUT "Is this acceptable? (y=yes)"; A$
190   IF A$<>"y" THEN T1=ABS(S-TT): GOTO 160
200 FOR I=1 TO 25
210   IF K(I)>0 THEN K(I)=-K(I)      'Mark this first pass.
220 NEXT I
230 GOSUB 1000
240 PRINT "Suggested time is";S
250 LINE INPUT "Is this acceptable? (y=yes)";A$
260   IF A$<>"y" THEN T1=ABS(S-TT): GOTO 230
270 PRINT "Schedule for both workers"
280 S1=0: S2=0
285 FOR I=1 TO 25
290   IF K(I)<0 THEN P=ABS(K(I)): PRINT T$(P), T(P): S1=S1+T(P)
300 NEXT I
310 PRINT "Sum, Worker 1:"; S1
330 FOR I=1 TO 25
340   IF K(I)>0 THEN PRINT T$(K(I)), T(K(I)): S2=S2+T(K(I))
350 NEXT I
360 PRINT "Sum, Worker 2:"; S2
```