How to Program

(An Informal Guide)

Chakravarthy Bhagvati

This is an informal guide to solving problems on a computer by writing programs. The emphasis is not so much on software engineering but on simple rules of thumb to make one a better programmer. *Follow these guidelines for **every programming assignment**, make it a habit and it is almost certain to halve the programming time while doubling the pleasure and returns*.

1 The three steps to writing good programs

Design: this is 60% of the process

Implementation: writing the program (the easiest part!)

Testing and debugging: after all, nobody's perfect

1.1 Design

Design itself is a four-step process:

1. *Generalize*

   Normally, one is given *an instance* of a problem and not the problem itself. It is very important to realize the distinction between the two. An instance is one specific example, many times a special case, of a more general problem. Generalization is done by expanding the type and range of inputs, identifying and expanding *hidden* parameters, and very rarely (only at the expert level) the problem itself into an equivalent family of problems.

2. *Enumerate specific cases*

   This is almost the opposite of Step 1. Reduce the general problem into specific subsets of problem instances or sub-problems. Identify and clearly list the limitations, ranges, parameters, etc. for each specific case. Make sure the original problem is included as at least one specific case :-)

3. *Make choices; give choices*

   It may be possible to solve only some of the cases identified in Step 2 easily and efficiently. One has to make certain decisions here: do you want to implement only those easy and efficient cases? do you want to make them the default option and provide the user with a choice to enable solving the more difficult cases?

4. *Organize*

   This often appears to be the hardest part but if you follow the above three steps, organizing the solution into clearly defined steps is rather straight-forward. The golden rule of thumb for organizing your solution: *separate the input and output aspects from the computation*.

Identify and group all necessary inputs. How many and what should be given by the users? What are better chosen as parameters (these are values that change only rarely; offer the ability to generalize; require expert knowledge, etc.)?

Computation is easy! Find a good algorithm, organize the computational module into major steps as described by the algorithm and *voila!* that's all.

Identify clearly the outputs and how they need to be presented to the users. Organize the outputs according to their importance. Let there be modules to summarize the results in addition to the raw values.

At the end of this step, the solution is broken down into a set of different modules or functions each with clearly defined inputs, functionality and outputs.

1.2 Implementation

Implementation is converting the design into a program. Once the design is well-done, implementation is easy. However, there are a set of choices to make at this stage.

The first is, how to get user input.

1. The program can prompt the user. I personally hate this one as it has many disadvantages: the user has to give all the inputs each time the program is run; it is easy for the user to make a mistake and it is not so easy to write user input routines to allow users to correct their mistakes; it makes the programs slow (users *are slow*, aren't they?!). My only advice is to remember what you are letting yourself into with this one and write your programs very carefully to handle mistakes. Nothing is worse than a program that dies because of wrong input from users.

2. The program can have command line options. This is my favourite — hey, I have been born and brought up on UNIXes. Many of the disadvantages disappear in this mode. More importantly, for me, it allows a later easy up-(sorry, down-) gradation to GUI. But, design your command line options well and make sure you write a good parser or use pre-written, well-tested code from the net.

3. Have a configuration file that contains all the user choices and have the input routines read the file. This is the best option if there are many inputs as the user can enter them once and it is also easy to correct mistakes. Parsing is also relatively easy compared to command line options.

4. Use a combination!

−1. **Finally, one thing you should NEVER DO: hard code input in the program**. You do this and I will pull you inside out, make you swallow yourself and dance on your remains with hob-nailed boots!! If not me, your supervisor, team leader, boss or *somebody* will, believe me.

As far as computation is concerned, use your knowledge of data structures and algorithms and write good code. In certain situations, I strongly recommend downloading well-tested code from the net.

How about the output? Again, the choices are somewhat similar to those for input. In many cases, it is fine to display output on the screen; leave sufficient white space, comment it well and keep things uncluttered. **Do not use functions to clear the screen, etc.** because they are unnecessary and make your code oh-so-Turbo-C-ish! If there is a lot of output, it may be good to write it into a file and display summary information on the screen.

Finally, an important point about output is to think about *how a user might use it*. For example, if the user is likely to plot the output, then it is a great idea to write the output into

a file in a format that allows for easy plotting by popular packages. In UNIX world, the output from one program is often sent as input to another and therefore it is not a bad idea to remove all unnecessary comments and write only the data out. If necessary, there may be an option to produce more human-readable output. By the way, this is another reason why I like command line options!

1.3 Testing and debugging

The under-rated and often despised aspect of programming — *to err is human, to debug, sub-human*. Books, tomes and treatises have been written on the subject. Industry swears by it, academicians swear *at* it. None of us is a God and we have to live with mistakes. So, it might be well to learn a few tricks to reduce the number of bugs before we get down to testing.

1.3.1 Where do most bugs occur?

1. *Reading input*

   One of the commonest mistakes early programmers make is to read input *as characters*. A typical example: the computer prompts, "Do you want to continue (y/n)? ", to which a learned (and an unusually generous) user responds with a "y" or an "n." What is wrong here? Nothing except that beginners read user input with `getc(), getch()` or `getchar()` functions. The result is a disaster. All I/O is buffered in most systems and a user has to type `<ENTER>` key to send the input to the computer and these character reading functions leave the `<ENTER>` keystroke behind for the *next `getc()`* — something that the programmer did not really want.

   One may think that the above can be fixed by judiciously adding dummy `getc()`s — **don't**. It won't work. The problem is that user input is a combination of real data *and whitespace*. While we can say that the user is stupid, make jokes and laugh at his/her expense when it comes to mistakes in the data, we cannot do that with whitespace. The user does not even know what it is.

   So what is the solution? **ALWAYS read user input as formatted data or strings using `scanf(), fscanf()` or best of all, `fgets()`. For airtight reading of user input, use `fgets()` followed by `sscanf()` and the like.**

2. *Uninitialized variables*

   Easiest to fix, **so do it, please!** These errors are extremely easy to detect. Run the program and if the answer gives you a 10- or 11-digit positive or negative number when the maximum or minimum you are expecting as an answer is only a couple of digits, there are uninitialized variables. Go through your code and simply initialize all variables at the time of their declaration. Instead of

   ```
   int i;
   ```

   use

   ```
   int i = 0;
   ```

3. *Uninitialized* pointer *variables*

   A special case of the above, but a very scary and unnerving one. This one gives the dreaded *segmentation violation*, *bus error* or something equally scary. It is not very hard to fix and does not occur frequently in languages more modern than 'C.'

There are only three ways to initialize a pointer variable — to `NULL` which does not solve the problem; assign an already allocated variable; or use `malloc()`, `calloc()`, etc. to really initialize it. Choose your poison and you are going to be fine.

4. *Forget arguments to functions*

   Oh boy, has this hit you yet? You use `printf()` to get extra information to identify a pesky bug and end up with a far worse output. The problem: you forgot to pass the variable for printing to `printf()`. In other words, you wrote

   ```
   printf(‘‘Value of var1: %d\n’’);
   ```

   instead of

   ```
   printf(‘‘Value of var1: %d\n’’, var1);
   ```

   This also occurs when calling functions. Again, a more modern language than ‘C’ flags these errors but in ‘C’ you only get warnings. And, as is the normal case, you ignored them, didn’t you? Pay attention to warnings — some of them are useful!

There are other bugs but they are much rarer these days. Logical bugs occur and are hard to fix but they don’t occur with the same frequency as above. A final advice is to learn using a *debugging tool*. You don’t need anything fancy, one that can tell you *where exactly a bug occurs* is enough. Once you know where the bug is, you don’t need a fancy-shmancy debugger to tell you *what it is*. Hey, we are smart enough, thank you!

1.3.2 Testing

The simplest way to test a program is to run it several times with different inputs. If the output is not one of the cases listed in the previous section, you’re most likely OK. Also, keep one realistic example worked out by hand so that you know the correct output for a given input. Verify that your program’s output matches your answer. If you are really a masochist, then you can try giving all kinds of wrong inputs just to see how your program handles them. Form a group of friends and test each other’s programs out. It is fun!

2 Solved Problem: Magic Square

Let us put all the points above into practice with this example.

> Write a program to create a 5×5 *magic square* using the numbers 1 – 25. A magic square is a square matrix in which the sum of numbers along any row, column or diagonal is the same.
> For example, the following is a 3×3 magic square with a magic sum of 15:
>
> | | | |
> |---|---|---|
> | 2 | 7 | 6 |
> | 9 | 5 | 1 |
> | 4 | 3 | 8 |

2.1 Design

Let us start with *generalizing* the problem given. What can we generalize? First, the size of the matrix: allow arbitrary size not just 5×5.

Second, how about the range of numbers? Can we use any set of N^2 consecutive numbers for an $N \times N$ magic square not just $1 \ldots N^2$. Even better, can any *arbitrary* set of numbers be arranged in a magic square?

Third, can we let the user specify a magic sum and then *derive* the other inputs such as the size and range of numbers from it?

Good, enough generalization (Lynn Truss[1], you may remove the comma too, if you please)! Let us now enumerate the specific cases:

1. Let the user specify the size, N. We arrange the numbers $1 \ldots N^2$ in a magic square.
2. Let the user specify the size N and a *starting number* S. We arrange the numbers $S, S + 1, \ldots, S + N^2$ in a magic square of size $N \times N$.
3. Let the user specify the size N and an *arbitrary sequence of numbers* $A_{1 \ldots N^2}$. We arrange the numbers in a magic square of size $N \times N$.
4. Let the user specify a magic sum M and a size N. We compute the required set of N^2 numbers and arrange them in a magic square.
5. Let the user specify a magic sum M. We compute the size N and the required set of numbers and arrange them in a magic square.

That's a lot of cases — enough for anyone to make and give choices. We will examine each case above. Take *Case 1*. When we scan the literature, we find that constructing magic squares is simple if N is *odd* but not so straight-forward if N is *even*. The only special case of $N = 4$ is easy. Therefore, let us choose *only to write the program for* odd *N*.

Cases 2 and 3 are related. The real question is, "*which sequence of numbers?*" If we scan the literature, we find that magic squares are always constructed from sequences of consecutive numbers although with arbitrary starting points. But the general question (i.e., *Case 3*) is interesting too, in a Martin Gardner[2] sense. These are now my own intuitions. Any sequence of numbers in *arithmetic progression* is fine. In fact, my belief is that a sequence of numbers that can be arranged in pairs such that the pairs have identical sums is sufficient but don't quote me on this. Anyway, let us be conservative and choose only to implement *Case 2*.

[1] The one who wrote that hilarious bestseller on punctuation titled, "*Eats, shoots and leaves*"

[2] This is the guy who is known for his brilliant mathematical puzzles and problems; and whose publications are a staple diet of *Scientific American*

Cases 4 and 5 are straight-forward if we restrict the required set of numbers to N^2 consecutive numbers starting with S. In *Case 4*, we need to compute only S while in *Case 5*, we somehow need to compute N and S.

How are magic sum, starting point and size related? Their relationship is given by the equation

$$M = N\left(S + \frac{N^2-1}{2}\right) \tag{1}$$

Given M and N,

$$S = \frac{M}{N} - \frac{N^2-1}{2}$$

Given that our problem is 5×5 magic square, let us make *that* the default. In other words, if the user gives no input, the program generates a 5×5 magic square using the numbers $1 \ldots 25$. The other cases are optional.

To structure the solution, separate the input, computation and output. Here is how the input is specified.

1. Default case: 5×5 magic square using numbers $1 \ldots 25$.
2. User specifies size as an odd number N.
3. User specifies size as an odd number N and a starting point S.
4. User specifies a magic sum M and size N.
5. User specifies a magic sum M only.

As far as the computation is concerned, we have only one case: constructing a magic square of odd size. One algorithm that we get from the literature is given in the box. You may want to verify that the 3×3 magic square given at the beginning of the section is constructed from the algorithm above.

At the end of the design phase, we have the following clearly defined modules/functions.

Input:

1. `getInputBySize(msSize, msStart)`

   This function prompts the user for size and starting point and returns them via the parameters. It returns 0 on success and any other value on failure.

2. `getInputByMagicSum(msSum, msSize, msStart)`

   This function prompts the user for magic sum and size, computes the other inputs and returns them via the parameters. It returns 0 on success and any other value on failure.

MAGIC SQUARES ALGORITHM FOR ODD N

1. Put the starting number S in the middle of the last column. Let current number $c = S$.
2. Repeat until all the cells are exhausted
   (a) Increment current number: $c = c + 1$.
   (b) Move upward and to the right of cell last filled by c, i.e., to cell $(i-1, j+1)$ if the last cell filled is (i, j).
   (c) There are five possible cases for the new location.
      i. If the cell is empty, put c there.
      ii. If the cell is already filled, move one cell to the left, i.e., to $(i-1, j)$.
      iii. If the column is invalid, i.e., $j > N$, then set $j = 1$. In other words, fill the first cell in that row.
      iv. If the row is invalid, i.e., $i < 1$, then set $i = N$. In other words, fill the last cell in that column.
      v. If both row and column are invalid, i.e., $i < 1, j > N$, then fill the cell $(i, j-1)$.

Computation: Computation is done by a single function which uses the algorithm above to fill the matrix `msMagicSquare`. The function

```
computeMagicSquare(msMagicSquare, msSize, msStart)
```

returns 0 on success and any other value on failure.

Output: Output is equally simple and done by a single function which writes out the computed magic square on the screen. The function

```
writeMagicSquare(msMagicSquare, msSize, msStart)
```

does not return any value.

2.2 Implementation

Implementation requires programming the above four functions. The user needs to input at most two values. Therefore, the input functions can read data directly from the user. The algorithm is also straight-forward and does not need any special data structures or tricks. My implementation is given in the appendix.

However, there are a few interesting implementation choices. The first is the way user decides between magic sum and magic square size. I decided to use command line option here. The usage of the program is

`magicsquares`**:** the default case where the program simply outputs a 5×5 magic square with numbers from 1 to 25.

`magicsquares bySize`**:** the user is prompted for the size and beginning of the range of numbers.

`magicsquares bySum`**:** the user is prompted for the magic sum and the program generates a corresponding magic square.

Any other call results in an error message that gives the correct usage.

The second interesting choice is the *bySum* option. When the user specifies a magic sum, the program uses Equation 1 with $S = 1$ and then finds N. For most magic sums, N found by substituting $S = 1$ will not result in an integer and therefore the size has to be separately input by the user. One observation from Equation 1 is that the magic sum *must be divisible by the size* and therefore the program attempts to find such sizes and provides the user with choices. The user may then choose a size and the program returns the appropriate magic square. Sometimes it may not be possible to construct a magic square for certain sums and the program says so. There are error checks to see that the program does not crash even if the user gives an impossible choice.

2.3 Testing

Testing is easy because we can verify if the output is a magic square by computing sums along the rows, columns and diagonals. Here are some inputs and outputs showing that the program is written correctly.

1. The default case: `magicsquares`

```
MagicSquare of Order 5x5 beginning at 1
-----------------------------------
|    9 |    3 |   22 |   16 |   15 |
-----------------------------------
|    2 |   21 |   20 |   14 |    8 |
-----------------------------------
|   25 |   19 |   13 |    7 |    1 |
-----------------------------------
|   18 |   12 |    6 |    5 |   24 |
-----------------------------------
|   11 |   10 |    4 |   23 |   17 |
-----------------------------------

-----   Magic Sum: 65   -----
```

2. Specifying the size: `magicsquares bySize`

```
Enter desired size of Magic Square (default 5): 7
Enter start of range of numbers in Magic Square (default 1):
MagicSquare of Order 7x7 beginning at 1
-------------------------------------------------
|   20 |   12 |    4 |   45 |   37 |   29 |   28 |
-------------------------------------------------
|   11 |    3 |   44 |   36 |   35 |   27 |   19 |
-------------------------------------------------
|    2 |   43 |   42 |   34 |   26 |   18 |   10 |
-------------------------------------------------
|   49 |   41 |   33 |   25 |   17 |    9 |    1 |
-------------------------------------------------
|   40 |   32 |   24 |   16 |    8 |    7 |   48 |
-------------------------------------------------
|   31 |   23 |   15 |   14 |    6 |   47 |   39 |
-------------------------------------------------
|   22 |   21 |   13 |    5 |   46 |   38 |   30 |
-------------------------------------------------

-----   Magic Sum: 175  -----
```

3. Sepcifying the size and start of the range: `magicsquares bySize`

```
Enter desired size of Magic Square (default 5): 9
Enter start of range of numbers in Magic Square (default 1): 12
MagicSquare of Order 9x9 beginning at 12
-------------------------------------------------------------
|   46 |   36 |   26 |   16 |   87 |   77 |   67 |   57 |   56 |
-------------------------------------------------------------
|   35 |   25 |   15 |   86 |   76 |   66 |   65 |   55 |   45 |
-------------------------------------------------------------
|   24 |   14 |   85 |   75 |   74 |   64 |   54 |   44 |   34 |
-------------------------------------------------------------
|   13 |   84 |   83 |   73 |   63 |   53 |   43 |   33 |   23 |
-------------------------------------------------------------
|   92 |   82 |   72 |   62 |   52 |   42 |   32 |   22 |   12 |
-------------------------------------------------------------
|   81 |   71 |   61 |   51 |   41 |   31 |   21 |   20 |   91 |
-------------------------------------------------------------
|   70 |   60 |   50 |   40 |   30 |   29 |   19 |   90 |   80 |
-------------------------------------------------------------
|   59 |   49 |   39 |   38 |   28 |   18 |   89 |   79 |   69 |
-------------------------------------------------------------
|   48 |   47 |   37 |   27 |   17 |   88 |   78 |   68 |   58 |
-------------------------------------------------------------

-----   Magic Sum: 468  -----
```

4. Specifying the magic sum: `magicsquares bySum`

```
Enter Magic Sum: 80
Enter desired size of Magic Square (Optimum 5):
MagicSquare of Order 5x5 beginning at 4
-----------------------------------
|   12 |    6 |   25 |   19 |   18 |
-----------------------------------
|    5 |   24 |   23 |   17 |   11 |
-----------------------------------
|   28 |   22 |   16 |   10 |    4 |
-----------------------------------
|   21 |   15 |    9 |    8 |   27 |
-----------------------------------
|   14 |   13 |    7 |   26 |   20 |
-----------------------------------

-----   Magic Sum: 80   -----
```

In this example, the program found that by substituting $S = 1$ in Equation 1, the largest magic square for which the magic sum is less than 80 is 5×5. As 80 divisible by 5, the program suggests the user to specify a size of 5.

5. Specifying the magic sum: `magicsquares bySum`

```
Enter Magic Sum: 2009
Enter desired size of Magic Square (Magic Sum 2009 should be divisible by size (~15)
Here are some possible choices: 7
Enter the desired size: 7
MagicSquare of Order 7x7 beginning at 263
-------------------------------------------------
|  282 |  274 |  266 |  307 |  299 |  291 |  290 |
-------------------------------------------------
|  273 |  265 |  306 |  298 |  297 |  289 |  281 |
-------------------------------------------------
|  264 |  305 |  304 |  296 |  288 |  280 |  272 |
-------------------------------------------------
|  311 |  303 |  295 |  287 |  279 |  271 |  263 |
-------------------------------------------------
|  302 |  294 |  286 |  278 |  270 |  269 |  310 |
-------------------------------------------------
|  293 |  285 |  277 |  276 |  268 |  309 |  301 |
-------------------------------------------------
|  284 |  283 |  275 |  267 |  308 |  300 |  292 |
-------------------------------------------------

-----   Magic Sum: 2009 -----
```

In this example, the program suggests 7 as a possible size because 2009 is divisible by 7 and no other number less than 15. 15 is the size computed by substituting $S = 1$ in Equation 1 and gives the largest magic square for which the magic sum is less than 2009.

6. Specifying impossible choices: `magicsquares bySum`

```
Enter Magic Sum: 230964
Enter desired size of Magic Square (Magic Sum 230964 should be divisible by size (~77)
Here are some possible choices: 57 19 3
Enter the desired size: 55
Please enter a size that divides 230964
Enter Magic Sum: 230964
Enter desired size of Magic Square (Magic Sum 230964 should be divisible by size (~77)
Here are some possible choices: 57 19 3
Enter the desired size: 3
MagicSquare of Order 3x3 beginning at 76984
----------------------
| 76985 | 76990 | 76989 |
----------------------
| 76992 | 76988 | 76984 |
----------------------
| 76987 | 76986 | 76991 |
----------------------

-----   Magic Sum: 230964       -----
```

7. Invalid choices: `magicsquares default`

```
Usage: magicsquares [bySum|bySize]
```

A Magic Square Program in 'C' Language

```
#include <stdio.h>
#include <math.h>

#define     MAXSIZE          100     /* Maximum size of the magic square */

/****** FUNCTION PROTOTYPES ******/
int getInputBySize(int *msSize, int *msStart);
int getInputByMagicSum(int *msSum, int *msSize, int *msStart);
int computeMagicSquare(int msMagicSquare[][MAXSIZE], int msSize, int msStart);
void writeMagicSquare(int msMagicSquare[][MAXSIZE], int msSize);

int main(int argc, char *argv[])
{
    int msOrder = 5, msStart = 1, msSum = 0, msOptionBySum = 0, rv = 0;
    int msMagicSquare[MAXSIZE][MAXSIZE], msDefaultOption = 1;

    if ((argc != 1) && (argc != 2)) {         /* Incorrect Usage */
       fprintf(stderr, "Usage: magicsquares <bySum>\n");
       exit(1);
    }
    if (argc == 2) {
       if ((strcmp(argv[1], "bySum") != 0) &&
           (strcmp(argv[1], "bySize") != 0)) {
         fprintf(stderr, "Usage: magicsquares [bySum|bySize]\n");
         exit(1);
    } else if (strcmp(argv[1], "bySum") == 0) {   /* By Magic Sum */
         msOptionBySum = 1;
         msDefaultOption = 0;
    } else                                        /* By Size */
         msDefaultOption = 0;
    }

    if (msDefaultOption == 0) {
       if (msOptionBySum == 0)
         do
              rv = getInputBySize(&msOrder, &msStart);
         while (rv != 0);
       else
         do
              rv = getInputByMagicSum(&msSum, &msOrder, &msStart);
         while (rv != 0);
    }

    if (computeMagicSquare(msMagicSquare, msOrder, msStart) != 0) {
        fprintf(stderr, "Error in computing magic square\n");
        exit(1);
    }
    writeMagicSquare(msMagicSquare, msOrder);     /* Output */
}
```

A.1 Input Functions

```
int getInputBySize(int *msSize, int *msStart)
{
     int rv = 0;
     char tmpBuffer[MAXSIZE];

     printf("Enter desired size of Magic Square (default 5): ");
     fflush(stdout);          /* To force output even without \n */

     if (fgets(tmpBuffer, MAXSIZE, stdin) != NULL) {
       if (sscanf(tmpBuffer, "%d", &rv) == 1) {
         if (rv % 2 == 0) {
              fprintf(stderr, "Desired size should be ODD\n");
              return (1);
         }
         if (rv <= 0) {
              fprintf(stderr, "Desired size > 0\n");
              return(1);
         }
         if (rv < MAXSIZE)
              *msSize = rv;
         else
              *msSize = MAXSIZE;
       }
     } else
       return(1);

     printf("Enter start of range of numbers in Magic Square (default 1): ");
     fflush(stdout);          /* To force output even without \n */

     if (fgets(tmpBuffer, MAXSIZE, stdin) != NULL) {
       if (sscanf(tmpBuffer, "%d", &rv) == 1) {
         if (rv <= 0) {
              fprintf(stderr, "Start should be >= 1\n");
              return (1);
         }
         *msStart = rv;
       }
     } else
       return(1);

     return(0);
}
```

```
int getInputByMagicSum(int *msSum, int *msSize, int *msStart)
{
     int rv = -1, firstGuess = 1, sqSize = (*msSize) * (*msSize), i = 0;
     int possible = 0;
     char tmpBuffer[MAXSIZE];

     printf("Enter Magic Sum: ");
     fflush(stdout);          /* To force output even without \n */

     if (fgets(tmpBuffer, MAXSIZE, stdin) != NULL) {
       if (sscanf(tmpBuffer, "%d", &rv) == 1) {
         if (rv <= 0) {
              fprintf(stderr, "Magic Sum should be >= 1\n");
              return (1);
         }
         *msSum = rv;
       }
     } else
       return(1);

     /******** Substitute S = 1 in Equaton 1 and estimate N ********/
     firstGuess = (int)(pow((double)(2.0 * *msSum), 1.0/3.0));
     if (firstGuess % 2 == 0)
       firstGuess += 1;

     if (*msSum % firstGuess == 0) {      /* Is Sum divisible by N */
       *msSize = firstGuess;
       printf("Enter desired size of Magic Square (Optimum %d): ",
           firstGuess);
     } else {
       printf("Enter desired size of Magic Square (Magic Sum %d ", *msSum);
       printf("should be divisible by size (~%d)\n", firstGuess);
       printf("Here are some possible choices: ");
       for (i=firstGuess - 1; i>2; i--) { /* Check divisibility */
         if (i % 2 == 1)                  /* Offer choice to users */
              if ((*msSum) % i == 0) {
                printf("%d ", i);
                possible = 1;
              }
       }
       printf("\n");
       if (possible == 0) {               /* If no number divides Sum */
         fprintf(stderr,
              "It is NOT POSSIBLE to build a magic square for %d\n",
              *msSum);
         return(1);
       }
       printf("Enter the desired size: ");
     }
     fflush(stdout);          /* To force output even without \n */
```

```
      if (fgets(tmpBuffer, MAXSIZE, stdin) != NULL) {
        if (sscanf(tmpBuffer, "%d", &rv) == 1) {
          if (rv % 2 == 0) {
               fprintf(stderr, "Desired size should be ODD\n");
               return (1);
          }
          if (rv <= 0) {
               fprintf(stderr, "Desired size > 0\n");
               return(1);
          }
          if (rv < MAXSIZE)                /* Accept user's choice and */
                 *msSize = rv;             /* build magic square with */
          else                             /* an approximate sum */
                 *msSize = MAXSIZE;
        }
      } else
        return(1);

      *msStart = *msSum / *msSize - ((*msSize) * (*msSize) / 2);
      if (*msStart < 1) {
        fprintf(stderr, "It is NOT POSSIBLE to construct a Magic Square ");
        fprintf(stderr, "with Sum %d and Size %d (%d)\n", *msSum, *msSize,
          *msStart);
        return(1);
      }

      return(0);
 }
```

A.2 Magic Square Computation

```
 int computeMagicSquare(int msMSq[][MAXSIZE], int msSize, int msStart)
 {
      int c = msStart, cpi = 0, cpj = 0;

      printf("MagicSquare of Order %dx%d beginning at %d\n",
          msSize, msSize, msStart);
      for (cpi = 0; cpi < msSize; cpi++)
        for (cpj = 0; cpj < msSize; cpj++)
          msMSq[cpi][cpj] = -1;

      /**** Starting Location in the middle of the last column */
      cpi = msSize / 2;
      cpj = msSize - 1;

      for (c = msStart; c < (msStart + msSize * msSize); c++) {
        msMSq[cpi][cpj] = c;                      /* Case i */
        cpi = cpi - 1;
        cpj = cpj + 1;
        if ((cpi < 0) && (cpj >= msSize)) {       /* Case v */
          cpi = cpi + 1;
```

```
          cpj = cpj - 2;
        } else {
          if (cpi < 0)                              /* Case iv */
               cpi = msSize - 1;
          if (cpj >= msSize)                        /* Case iii */
               cpj = 0;
        }
        if (msMSq[cpi][cpj] != -1) {                /* Case ii */
          cpi = cpi + 1;
          cpj = cpj - 2;
        }
      }

      return(0);
}
```

A.3 Output Function

```
void writeMagicSquare(int msMSq[][MAXSIZE], int msSize)
{
      int i, j;

      for (i=0; i<msSize; i++) {
        for (j=0; j<msSize; j++)
          printf("-------");
        printf("-\n");
        for (j=0; j<msSize; j++)
          printf("| %4d ", msMSq[i][j]);
        printf("|\n");
      }
      for (j=0; j<msSize; j++)
        printf("-------");
      printf("-\n");

      printf("\n-----\tMagic Sum: %d\t-----\n",
          msMSq[msSize/2][msSize/2] * msSize);
}
```