

Assignment 4
C Programming, 202-1-9081
Programming I, 202-1-9011
Fall Semester 2003/4

17th December 2003

Abstract

The purpose of the assignment is to practice solving algorithmic problems using recursion, in C language.

Submission deadline is **December 29, 2003** (midnight). All questions about this assignment should be directed to **Michael** (orlovm@cs.bgu.ac.il).

1 Magic Squares

A *magic square* is an arrangement of the numbers from 1 to n^2 in an $n \times n$ matrix, with each number occurring exactly once, and such that the sum of the entries of any row, any column, or any main diagonal is the same. For example, Fig. 1 shows a 3×3 magic square.

8	1	6
3	5	7
4	9	2

Figure 1: A 3×3 magic square. All rows, columns and diagonals sum to 15.

So far, so simple... You will write a program that will generate a square using definition similar to the one above, but more generalized.

2 Generalized Magic Squares

The entries in a *generalized magic square* are integer $\langle h, v, d \rangle$ triples, where for the purpose of summation, the triple is considered as

- h , when summing a row (horizontally);
- v , when summing a column (vertically);
- d , when summing either of two main diagonals.

There are no restrictions on the values of $\langle h, v, d \rangle$ entries (such as the full $1, \dots, n^2$ range for the regular magic squares).

Of course, a generalized magic square is only magic if all rows, columns and diagonals sum to the same value. An example of such square is given in Fig. 2.

$\langle 5, 6, -5 \rangle$	$\langle 3, 1, 6 \rangle$	$\langle 10, 9, 3 \rangle$
$\langle 2, 7, 4 \rangle$	$\langle 3, 9, 10 \rangle$	$\langle 13, 9, 8 \rangle$
$\langle 7, 5, 5 \rangle$	$\langle -4, 8, -17 \rangle$	$\langle 15, 0, 13 \rangle$

Figure 2: A 3×3 generalized magic square. All rows, columns and diagonals sum to 18.

In order to better understand why a generalized magic square is considered “magic”, it can be easier to think about matrix elements as triangles, as shown in Fig. 3.

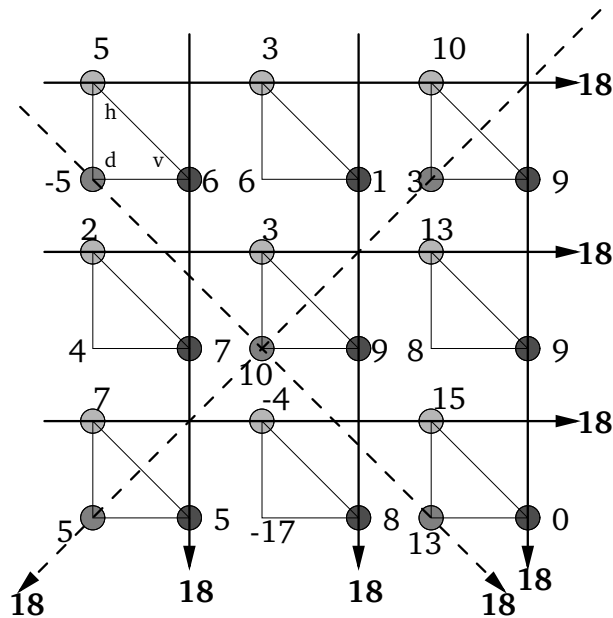


Figure 3: An equivalent graphical representation.

3 Input/Output Specification

The program should accept as its input $n > 0$ (the size of the square), number of requested solutions $k \geq 0$, and n^2 triples $\langle h_1, v_1, d_1 \rangle, \dots, \langle h_{n^2}, v_{n^2}, d_{n^2} \rangle$. The input is guaranteed to be correct, so no special verification is necessary.

After this the program should print h, v and d values separately, as shown in the example.

The program should then output the k first magic squares found, i.e., when descending down the recursion tree, the matrix entries are considered ordered

3 INPUT/OUTPUT SPECIFICATION

left-to-right, then top to bottom (entries of the first row, then entries of the second row, and so on); also, values given first in the list of $\langle h_i, v_i, d_i \rangle$ triples are tried first. This means that in case of success, the output of your program should be exactly the same as the one from the sample executable (number of spaces may vary, and you can avoid indenting).

The program should then print the number of found solutions (as in the examples). If after printing k' solutions, no more magic squares are possible ($0 \leq k' < k$), the program should also print a single-line message containing the word *fail* somewhere. In either case, the program should quit immediately afterwards.

Each triple is printed as (a,b,c), with no spaces inside it, except for the case where $a = b = c$ — then the triple is printed as a single number, without parentheses.

Four examples follow (text in *italic* is user input)...

```
----- Successful run on 3 x 3 matrix -----
Matrix size: 3

Solutions number: 1

Triple 1: 3 9 10
Triple 2: 15 0 13
Triple 3: 3 1 6
Triple 4: 7 5 5
Triple 5: 13 9 8
Triple 6: -4 8 -17
Triple 7: 5 6 -5
Triple 8: 2 7 4
Triple 9: 10 9 3

h values: [3, 15, 3, 7, 13, -4, 5, 2, 10]
v values: [9, 0, 1, 5, 9, 8, 6, 7, 9]
d values: [10, 13, 6, 5, 8, -17, -5, 4, 3]

(15,0,13)    (-4,8,-17)    (7,5,5)
(13,9,8)     (3,9,10)     (2,7,4)
(10,9,3)     (3,1,6)      (5,6,-5)

Found 1 solution.
```

```
----- Successful run on 4 x 4 matrix -----
Matrix size: 4

Solutions number: 10000000

Triple 1: 1 1 1
Triple 2: 2 2 2
Triple 3: 3 3 3
Triple 4: 4 4 4
Triple 5: 5 5 5
Triple 6: 6 6 6
Triple 7: 7 7 7
```

```

Triple 8: 8 8 8
Triple 9: 9 9 9
Triple 10: 10 10 10
Triple 11: 11 11 11
Triple 12: 12 12 12
Triple 13: 13 13 13
Triple 14: 14 14 14
Triple 15: 15 15 15
Triple 16: 16 16 16

h values: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
v values: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
d values: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]

1          2          15          16
12         14         3           5
13         7          10          4
8          11         6           9

1          2          15          16
13         14         3           4
12         7          10          5
8          11         6           9

:

16         15         2           1
5          3          14          12
4          10         7           13
9          6          11          8

Found 7040 solutions.
Failed to find 9992960 magic squares.

```

Failed run on 2×2 matrix

```

Matrix size: 2

Solutions number: 10

Triple 1: 1 1 1
Triple 2: 2 2 2
Triple 3: 3 3 3
Triple 4: 4 4 4

h values: [1, 2, 3, 4]
v values: [1, 2, 3, 4]
d values: [1, 2, 3, 4]

Found 0 solutions.

```

```
Failed to find 10 magic squares.
```

```
                                Hmmmm...  
Matrix size: 2  
  
Solutions number: 0  
  
Triple 1: 1 1 1  
Triple 2: 2 2 2  
Triple 3: 3 3 3  
Triple 4: 4 4 4  
  
h values: [1, 2, 3, 4]  
v values: [1, 2, 3, 4]  
d values: [1, 2, 3, 4]  
  
Found 0 solutions.
```

4 Guidelines

Start with writing the input handling part. It is best to store the triples in separate arrays, one for the h -values, one for the v -values, and one for the d -values.

Then, define a matrix in which you keep *indexes* into these arrays. Since you can't move values between different triples, these indexes are always the same, so you keep a single index in each matrix cell. (It is ok to keep indices instead of indexes).

Now, the search proceeds in recursive fashion, when the maximum tree depth is n^2 . Upon each recursive entry, you choose one index which wasn't used before. **You should also check that the sums are still ok.** The checking occurs only if the new index terminates a row (column, diagonal), and only for that row (column, diagonal). Otherwise, you will make too many checks.

Once you reach recursion depth of n^2 , you have a solution!

There's something I didn't mention... How do you know which sum is correct? You can initialize the target sum with the sum of the first row, and then try to make all the other rows (columns, diagonals) sum to this sum. But it is better to point for the correct sum from the start. How to do this? It's quite easy, try to think about the total sum of the elements' values, and its relation to the target sum.

But you will then ask, which values, h -values, or v -values? Both, of course... And if they aren't equal? Then there's no solution! Add this check to the program.

In short, like I always say, make sure you understand the solution *process* before coding it. Writing the program should be the easiest part of successfully completing the assignment.

5 Submission

Please, do not cheat! We routinely use advanced similarity checking software.

Points will be taken down if solutions order is not according to the “standard” order in the recursion tree, as described in Sec. 2.

You will receive a bonus if your program completes the second (or similar) example in under 15 minutes on modern computer.

And, as the old song goes,

When I find my code in tons of trouble,
Friends and colleagues come to me,
Speaking words of wisdom:
“Write in C.”

As the deadline fast approaches,
And bugs are all that I can see,
Somewhere, someone whispers:
“Write in C.”

Write in C, Write in C,
Write in C, oh, Write in C.
BASIC’s dead and buried,
Write in C . . .

בהצלחה!