< Prev	Recursion Index	Next >

Magic Squares

In this page, I'm going to show you the permutation-capabilities of Recursion. Permutation means a combination of certain units in all possible orderings. Recursion can be effectively used to find all possible combinations of a given set of elements. This has applications in anagrams, scheduling and, of course, Magic Squares. And if you're interested, Recursion can also be used for cracking passwords. ;-)

First, what *is* a magic square?
A magic square is a 'matrix' or a 2-dimensional grid of numbers. Take the simple case of a 3x3 magic square. Here's one:-

4	3	8
9	5	1
2	7	6

A Magic Square contains a certain bunch of numbers, in this case, 1..9, each of which has to be filled once into the grid. The 'magic' property of a Magic Square is that the sum of the numbers in the rows and columns and diagonals should all be same, in this case, 15.

Try making a 3x3 magic square yourself. It's not that easy. If it *was* easy, try a 4x4 grid with numbers 1..16. And what about 5x5, 6x6...? That's where computers come in! Okay, now, how do we go about programming something *we* hardly understand. The answer is : *brute force*. The computer may not be smart, but it can certainly do something you would take months to do, and that is, pure calculations - millions and millions of them. To have an analogy, suppose you are given two identical photos. It would take you just a glance of the eye to agree that they're same. The computer, on the other hand, would compare them dot by dot, pixel by pixel, and then agree that they're same. In the process, both you and the computer accomplish the same task in pretty much the same time, but in very different ways. So, the answer is brute force.

Okay, we've got a bunch of numbers. We've got to arrange them in a matrix so that the sum is equal in all directions.

Since we don't have a clue about any strategy, we'd just have to..... Try All Possibilities! Yes, and that's what a computer is meant to do - brute force.

Okay, try figuring it out yourself at this stage. You know everything there is to know - the concept of magic squares, the need to check all possibilities, and that the answer lies in recursion. To help ease the complexity, you can make a simple function (besides the *recursive* function) to check if a square is magical. So, basically, you've got to try all possibilities and permutations, and for each one, you have to call the function to test if the square is magical. I seriously suggest you stop at this point, and brainstorm to extremes. Better yet, just finish the program. It isn't that tough.

Given up? Fine. Having said all that I said, only the heart of the program is left to explain. The question boils down to, how do we accomplish permuting a set of numbers (or objects) using recursion? For simplicity sake, we work with a 1-dimensional array. In the case of a 3x3 square, let's have a 9-length single-dimensional array. So we have numbers 1 to 9 and 9 locations to put them into. The permutations would be:-

```
123456789
123456798
123456879
123456897
123456978
123456987
....
....
987654321
```

Conversion from single to 2-D array is fairly simple, and may be done within the magic-testing function that we shall call 'TestMagic'.

As a preliminary exercise, try to program the following sequence...

```
111,112,113,121,122,123,131,132,133,
211,212,213,221,222,223,231,232,233,
311,312,313,321,322,323,331,332,333.
```

...using For loops.

Answer: It's as simple as:-

```
for i = 1 to 3
    for j = 1 to 3
        for k = 1 to 3
            print i,j,k
```

Now, try it using recursion.

Answer: Think of it this way: *i* loops from 1 to 3. For every value of *i*, *j* loops from 1 to 3. For every value of *j*, *k* loops from 1 to 3. For every value of *k*, the values are displayed. So, basically, these three loops perform very similar functions, which can therefore be reduced to a single recursive function with a single loop.

```
void Func(n)
{
    for i = 1 to 3
    {
       A[n] = i
       if (n<3)
            Func(n+1)
        else
            print A[1],A[2],A[3]
    }
}
```

A[] is simply an array of integers. The function should be invoked with initial `n` value as 1, i.e., `Func(1)`. Trace the program to figure out the output.

For each pass through the loop, the function's child's loop is completely executed, by means of the recursive call. [A *child* of a given instance is the instance that it creates and calls.]

Just in order to confuse you even more, the same function can be written in this way:-

```
void Func(n)
{
    if (n<4)
    {
        for i = 1 to 3
        {
            A[n] = i
            Func(n+1)
        }
    }
    else
        print A[1],A[2],A[3]
}
```

Anyway, coming back to our magic squares and our 9-length array which is to be permuted....
We can easily adopt the above functions, with one difference: the numbers should not repeat. If the numbers *were* allowed to repeat, we could easily write:

```
void Permute(n)
{
    for i = 1 to 9
    {
        A[n] = i
        if (n<9)
            Permute(n+1)
        else
            TestMagic( )
    }
```

```
}
```

A[] is the 9-length array.
The function should be called with n=1 initially. Function TestMagic checks whether the square represented by the array is magical, and displays it if it is so.

However, we must not allow repetition, as per the rules and regulations of magic square authorities and environmental agencies worldwide. There could be several ways to perform this check, and it is left open to you. What *I* did was: I kept another 9-length array, which contained information about which number was used, and which was not.
E.g.: Say B[] is the extra array. If B[2]=0, then, number 2 is still free. If B[2]=1, then, number 2 is already used. Whenever we 'go to' a recursive call and 'come back' from a recursive call, we should update this array. One possible algorithm could be:-

```
void Permute(n)
{
    for i = 1 to 9
        if B[i]=0 // if i is free
        {
            B[i]=1
            A[n]=i
            if n<9
                Permute(n+1)     //recurse
            else
                TestMagic( )    //at the end
            B[i]=0
        }
}
```

Note the way I set and reset the value of B[i]. For all deeper recursive instances of the function, value of B[i] is 1. And when we come back, we undo this change. Strictly speaking, the same concept is used while changing the value of variable n. The value of n sent to the child instance is different from the value of n in the current instance. And upon returning from the child instance, the value of n is reset to its old value. (This is taken care of automatically by the passing of parameters between the calls, so we don't have to bother explicitly changing the value of n.)

The rest is left to you. Try the program, and taste the results yourself. There are totally 8 solutions for the 3x3 matrix with numbers 1..9. Did you get them all?

We have worked out this problem by realizing that we need to find all permutations of the numbers 1..9, and for each permutation, check if it satisfies the condition. There's another way of looking at the same thing, which is a common viewpoint when talking about recursion. It is called '*BackTracking*'. In effect, what our program does is, when it finds (at the 9th recursion) that a square is not magical (say),

it 'goes back' to the previous instance of the function, and even more back if needed. This is called BackTracking and is used in all recursive-problem-solving programs.

Even if you own a 800MHz m/c, you may have noticed that the calculation took at least a few seconds, maybe longer. This is because the algorithm is not very efficient. It is possible to get all the results in under a second on *any* machine, if we tweak this algorithm here and there. That's what we'll do next.

Tweak

If you observe (and imagine) a bit... okay, more than just a bit.... you will realize that a LOT of the possibilities we test are actually quite useless. The first state is:

1	2	3
4	5	6
7	8	9

or, 1 2 3 4 5 6 7 8 9.

Now, the begins by permuting 8-9, then 7-8-9, then 6-7-8-9... which all takes time. All these are useless until we start permuting the top row, since 1+2+3=6, but we need 15. Get it? It's going to take a LOT of permutations before we finally backtrack to 3,2 and 1, which remain as sum 6 all this while. So, this 1-2-3 combination really sucks, and we should never allow it.

I wouldn't be describing this problem if I didn't know the answer, so here it is: While filling the numbers, *at every row*, we check if the sum is 15. So, before we go on to the second row (or the 4th element in our 1-D array) we check the sum of the first row - if it isn't 15, go back and permute... until the sum is 15. So, now the first row has a sum 15 (done very quickly since we permute only 3 locations) and all those useless permutations of the other two rows are not attempted. The same should happen for the second row, which saves some time, but not as much as that for the first row. Finally, after all 9 elements are inserted, check the columns and diagonals.

Actually what happens is, if the sum of the first row elms is *not* 15, it backtracks until the sum is 15. If the sum is now 15, it 'goes forward' and checks the second row for sum 15. If second row sum is not 15, it backtracks to try to get the sum as 15. If all permutations within row 2 are complete, it backtracks to row 1... The following function checks the sum at every row.

```
void Permute(n)
```

```
{
    for i = 1 to 9
        if B[i]=0     // if i is free
        {
            B[i]=1
            A[n]=i
            if (n % 3 = 0) // if n is div by 3
            {
                if RowSum() = 15
                {
                    if n<9
                        Permute(n+1)
                    else   // at the end
                        TestMagic()
                }
            }
            else
                Permute(n+1) //recurse
            B[i]=0
        }
}
```

Tweak 2

Even the above program is a bit wasteful. Consider one of the cases in which the above program successfully fills the entire 9 locations (The row sums are all 15):-

1	5	9
2	6	7
3	4	8

This is by no means a magical square. Only the row sums are proper. What about the column sums? Look at column 1. Isn't it a little familiar? Right, we're back to the same old philosophies - this time, for the columns. A little thinking (for a guy like Einstein) or a lot of it (for you and me) will convince you that we also need to check and maintain *column* sums (as 15) along the way. So, we need to check both rows and columns. A little More thinking will convince you (poorly) that we need to fill the rows and columns alternatively to maximize efficiency (and confusion). At this point, keep in mind one of the greater truths that programmers (like you and me) live by: "*The more complicated your program, the more you are respected*." There are at least 2 good ways to alternate rows and columns.

	1	2	3
1	**a**	**b**	**c**
2	**d**	**e**	**f**
3	**g**	**h**	**i**

The first way:

row1, column1, row2, column2, row3,column3.
or,
a-b-c-d-g-e-f-h-i.
At c, check row1 sum. At g, check column1 sum. At f, check row 2 sum. At h, check column2 sum. At i check row3 and column3 sum and diagonals as well. So, with a very few 'fillings' we can extract the solutions.

The second way: In a spiral pattern:
row1,column3, row3(reverse), column1(rev), row 2, column2
or,
a-b-c-f-i-h-g-d-e.
At c, check row1. At i, check column3. At g, check row3. At d, check column1. At e, check row2, column2 and diagonals. And, there you have it. The solutions will pop out in a second.

Exercises

You should be able to solve these problems:-

1. Write a function using Recursion to display all anagrams of the word 'RECURSION'.

2. Write a function using Recursion to display all anagrams of any string entered by the user.

3. Write a function using Recursion to display all anagrams of a string entered by the user, in such a way that all its vowels are located at the end of every anagram. (E.g.: Recursion => Rcrsneuio, cRsnroieu, etc.) Optimize it.

4. Write a function using Recursion to do the following: You have 20 different cards. You have only 7 envelopes. You can therefore send only 7 cards. Display all possible ways you can fill the 7 envelopes. (Every card is different)

5. Same as above, except: Of the 20 cards, 5 are identical birthday cards, and 7 others are identical anniversary cards. Find all permutations, without repetitions. (Swapping two identical cards doesn't create a new permutation.)

6. Write a function using Recursion to crack a password. The password is of unknown length (maximum 10) and is made up of capital letters and digits. (Store the actual password in your program, just for checking whether the string currently obtained is the right password.)

7. Write a function using Recursion to do the following: You are the manager in a small factory. You have 7 workers and 7 jobs to be done. Each worker is assigned one and only one job. Each worker demands different pay for each job. (E.g.:

Worker Sam demands $10 for welding, $15 for carpentry, etc. Worker Joe demands $12 for welding, $5 for carpentry, etc.) Find the job-assignment which works out cheapest for you.

erw_nerve@email.com *July 2000*

Comments:

This page is located at http://personal.vsnl.com/erwin/magic.htm

PANIC

Display thank-you page Return to this page Submit

Recommend this page to someone?

e-mail

e-mail