CISS350: Data Structures and Advanced Algorithms
Assignment 1

Objectives

1. Review of CISS240/CISS245 including variables, input/output, branching, loops, arrays, pointers, functions.

The format of your program must look like this:

```
// Name: smaug
// File: a01q01.cpp

#include <iostream>

int main()
{
    *** YOUR WORK HERE ***
    return 0;
}
```

replacing "smaug" with your name. In particular:

1. You must have your name and the name of the file at the top of each C++ source file as shown above.
2. The last thing printed must be a newline.

Read the questions carefully before diving in.

Note that you should create a new project for each question. For easy maintenance of your assignments, I suggest you have a folder `ciss350` somewhere in your `Documents`, and in that you have a folder `a`, and in folder a you have a folder `a01`, and you have solutions folders `a01q01`, `a01q02`, etc. in the folder `a01`:

```
.
.
.
ciss350
|
+- a
```

```
|
+- a01
   |
   +- a01q01
   |
   +- a01q02
```

All the relevant files (cpp and header files) for question 1 must be in folder `a01q01`. Etc.

Some test cases are included in the problems. You are strongly advised to add more test cases on your own.

Q1. [Spiral] ASCII art. (Do not use arrays.)

The test cases explains what you need to do. Make sure the drawing is done by a function called with the following prototype:

```
void draw_spiral(int);
```

In other words, the skeleton code is

```
#include <iostream>

void draw_spiral(int n)
{
}

int main()
{
    int n;
    std::cin >> n;
    draw_spiral(n);

    return 0;
}
```

Test 1

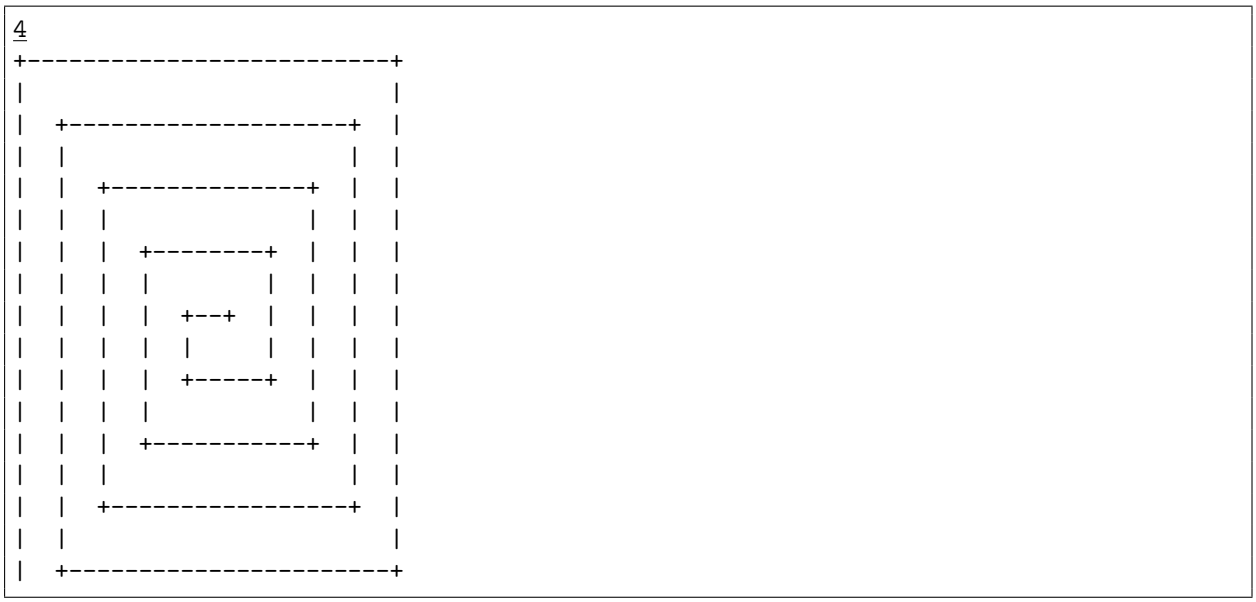
```
1
+--------+
|        |
|  +--+  |
|  |     |
|  +-----+
```

Test 2

```
2
+--------------+
|              |
|  +--------+  |
|  |        |  |
|  |  +--+  |  |
|  |  |     |  |
|  |  +-----+  |
|  |           |
|  +-----------+
```

Test 3

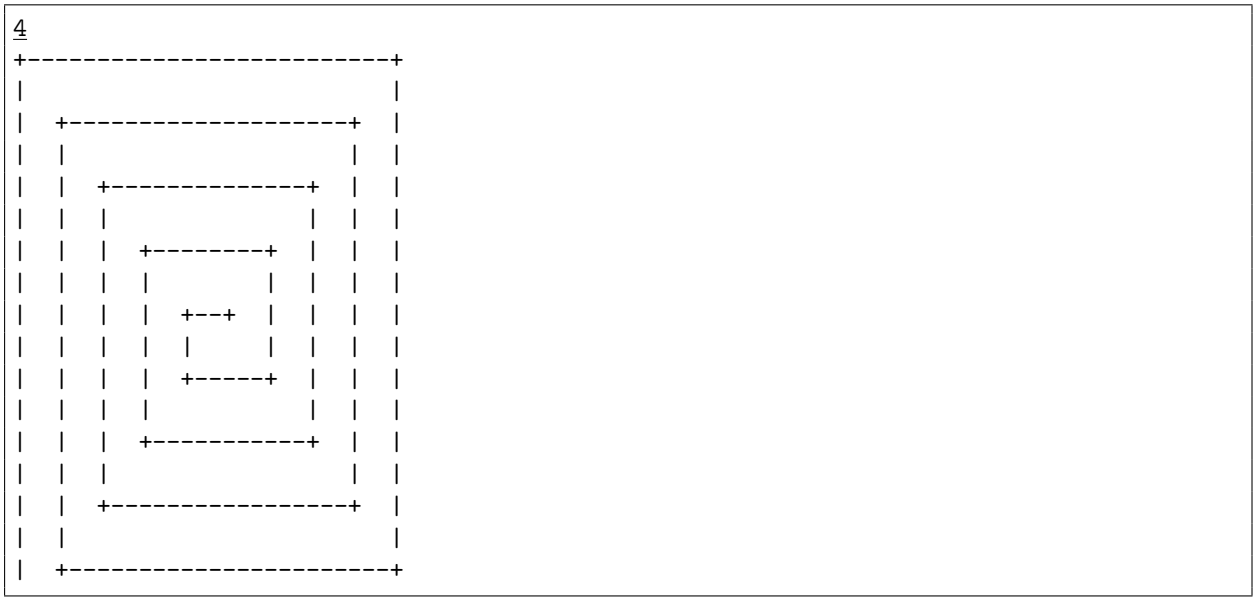
Test 4



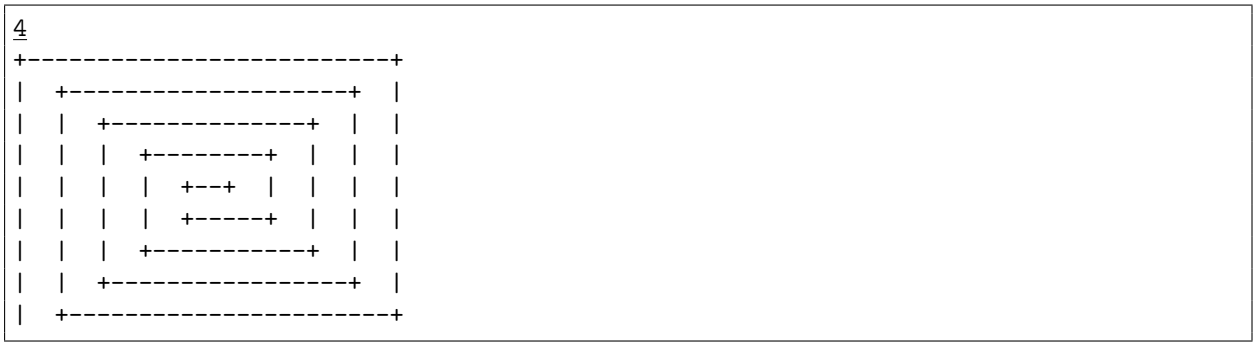
WARNING: ... INCOMING SPOILERS ... Hints on next page. Use only if necessary.

Hints

Look at the case when the input is 4:



Notice that the first line of output is "more similar" to the third and fifth than the second and fourth. So if you look at the 1-st, 3-rd, 5-th, ... lines of output the picture looks like this:



In other words the pseudocode should look like this:

```
for i = 1, 2, 3, 4, ..., 17:
    if i is odd:
        draw it in a certain way
    else:
        draw it in another way
```

This is similar to the problem of drawing alternativing characters, something like

```
5
*
@
*
@
*
```

Going back to our problem, I suggest you focus on the odd case first, i.e.,:

```
for i = 1, 2, 3, 4, ..., 17:
    if i is odd:
        draw it in a certain way
```

Of course later you have to make the program work for any input n. When the input is `4`, the number of lines printed is `17`. You have to experiment to see what happens when the input is 1, 2, 3, 5. You should be able to figure out a formula for the number of lines printed in terms of the user input n (look at the `??` below):

```
for i = 1, 2, 3, 4, ..., ??:
    if i is odd:
        draw it in a certain way
```

If you look at Test 1, Test 2, Test 3, Test 4, you'll see that

- when $n = 1$, the number of lines printed is 5.
- when $n = 2$, the number of lines printed is 9.
- when $n = 3$, the number of lines printed is 13.
- when $n = 4$, the number of lines printed is 17.

So what is `??` in terms of n?

The output lines of the top half are similar but slightly different from the bottom half. The top half looks like this:

```
4
+-------------------------+
|  +--------------------+  |
|  |  +--------------+  |  |
|  |  |  +--------+  |  |  |
|  |  |  |  +--+  |  |  |  |
```

and the bottom half looks like this:

```
4
|  |  |  |  +-----+  |  |  |
|  |  |  +-----------+  |  |
|  |  +-----------------+  |
|  +-----------------------+
```

So the pseudocode should look like this:

```
for i = 1, 2, 3, 4, ..., ??:
    if i is odd:
        if i < 10:
            draw it in a certain way (for top half)
        else:
            draw it in a certain way (for bottom half)
```

I suggest you focus on the top half first, i.e.,

```
for i = 1, 2, 3, 4, ..., ??:
    if i is odd:
        if i < 10:
            draw it in a certain way (for top half)
```

Here's the output again for the top half when `i` is odd – I've included the value of `i` on the left:

```
i
1  +-------------------------+
3  |  +--------------------+  |
5  |  |  +--------------+  |  |
7  |  |  |  +--------+  |  |  |
9  |  |  |  |  +--+  |  |  |  |
```

You see that for each value of `i` in the above, you have to

- print a bunch of `"|  "`,
- followed by `'+'`,
- followed by a bunch of `'-'`,
- followed by `'+'`,
- followed by a bunch of `"  |"`.

So the pseudocode now becomes:

```
for i = 1, 2, 3, 4, ..., ??:
    if i is odd:
        if i < 10:
            draw a bunch of "|  "
            draw '+'
            draw a bunch of '-'
            draw '+'
            draw a bunch of "  |"
            draw newline
```

In fact it's easy to count the number of times to draw the things in the bunches:

```
i
1  +-------------------------+    0"|  ", 1'+', 26'-', 1'+', 0"  |", 1'\n'
3  |  +--------------------+  |    1"|  ", 1'+', 20'-', 1'+', 1"  1", 1'\n'
5  |  |  +--------------+  |  |    2"|  ", 1'+', 14'-', 1'+', 2"  1", 1'\n'
7  |  |  |  +--------+  |  |  |    3"|  ", 1'+',  8'-', 1'+', 3"  1", 1'\n'
9  |  |  |  |  +--+  |  |  |  |    4"|  ", 1'+',  2'-', 1'+', 4"  1", 1'\n'
```

You now have to relate the values of `i` = 1,3,5,7,9 to the number of `"|  "` to draw:

```
i
1  +-------------------------+    0"|  "
3  |  +--------------------+  |    1"|  "
5  |  |  +--------------+  |  |    2"|  "
7  |  |  |  +--------+  |  |  |    3"|  "
9  |  |  |  |  +--+  |  |  |  |    4"|  "
```

Do you see that: (9 - 1)/2 = 4, (7 - 1)/2 = 3, etc.

```
for i = 1, 2, 3, 4, ..., ??:
    if i is odd:
        if i < 10:
            draw (i - 1)/2 "|  "
            draw '+'
            draw a bunch of '-'
            draw '+'
            draw a bunch of "  |"
```

In other words

```
for i = 1, 2, 3, 4, ..., ??:
    if i is odd:
        if i < 10:
            for k = 1, 2, 3, ..., (i - 1)/2:
                draw "|  "
            draw '+'
            draw a bunch of '-'
            draw '+'
            draw a bunch of "  |"
```

The above should get you going.

Q2. [Rising hills] ASCII art. (Do not use arrays.)

The test cases explains what you need to do. Make sure the drawing is done by a function with the following prototype:

```
void draw_rising_hills(int);
```

Test 1

```
1
*
```

Test 2

```
2
  *
****
```

Test 3

```
3
      *
  *  ***
*********
```

Test 4

```
4
            *
      *    ***
  *  ***  *****
***************
```

Test 5

```
5
                  *
            *     ***
      *    ***    *****
  *  ***  *****  *******
************************
```

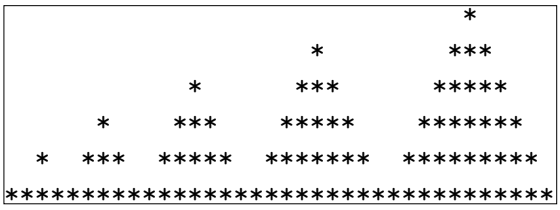
Test 6

```
6
                                *
                    *          ***
              *       ***       *****
       *     ***     *****     *******
   *   ***   *****   *******   *********
***********************************
```

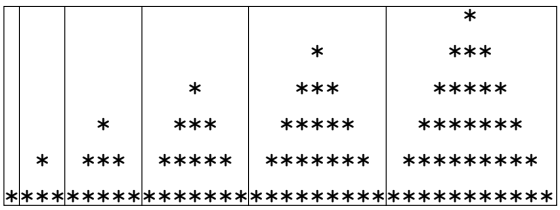
WARNING: ... INCOMING SPOILERS ... Hints on the next page.

Hints

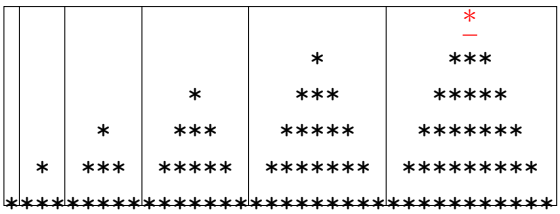
Look at the test case when the input is $n = 6$:



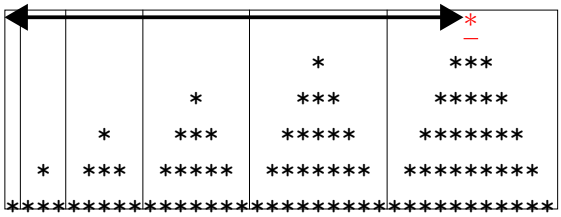
You might want to think of this:



Why is this helpful? Because if you want to draw the first star (at the first line of output):



you would need to print a bunch of spaces:



How many spaces? For the case when the user input is $n = 6$, the number of spaces is the base of the rectangles containing the hills before the largest hill #6, plus roughly half of the base of hill #6. Right? The first hill has base of length 1, the second hill has base of length 3, the third hill has base of length 5, etc.

For the printing of the second line of output, you will also need to use roughly the same idea when

you print the top of the second-to-last hill.

Q3. Write a program that accepts and prints the prime factorization of the number. (Do not use array.) [You definitely want to look at the review problem set as preparation for this problem.]

Test 1

```
1
1
```

Test 2

```
2
2
```

Test 3

```
3
3
```

Test 4

```
4
2^2
```

Test 5

```
5
5
```

Test 5

```
6
2 * 3
```

Test 6

```
8
2^3
```

Test 7

```
9
3^2
```

Test 8

```
10
2 * 5
```

Test 9

```
12
2^2 * 3
```

Test 10

```
100
2^2 * 5^2
```

Test 11

```
1265265
3^2 * 5 * 31 * 907
```

WARNING: ... INCOMING SPOILERS ... Hints on next page. Use only if necessary.

Hints

The following is meant to help you think through the problem-solving problem, specifically to show you one possible path to breakdown the problem. The pseudocode (or ideas) is pseudocode and is only meant to point you in certain directions. It's not meant to be complete – you have to add to it and modify it rather than to use it blindly. Also, after you have solved some smaller problems, sometimes when you glue two smaller solutions together, you will realize that one of the smaller solutions might have to be changed slightly.

OK, here we go ...

As mentioned before, you always try to find smaller problems within your problem and solve those smaller problems.

Looking at this:

```
1265265
3^2 * 5 * 31 * 907
```

The above prints a bunch of prime-powers dividing the user input n. In the above the first prime-power is 3^2. If you think about it, you should at least print the first prime-power. (In other words, instead of printing 4 prime-powers, print the first prime power. Why? Because if you can't print the first prime power, you can't possibly print 4 of them.)

To print the first prime-power dividing n, you have to at least print the smallest prime dividing n. In other words, in the above example, instead of computing 3^2, you should try to compute 3.

Based on the above breakdown of the problem, this is what I would do first: Get a positive integer from the user and print the first prime dividing the integer.

I will create about 5 test cases for this problem. (Yes, it's perfectly OK to create test cases *before* you even design the pseudocode. This is called test-driven development. This is what actually happens in real-life in situations like dating, buying a house, etc.)

The first task looks like this:

```
get n from user
print first prime dividing n
```

Getting n from the user is easy. For the second statement, of course you can't tell C++ to give you the first prime. So you have to translate that to something C++ understands. The smallest prime is 2 (1 is not a prime). So you would have to do something like this:

```
get n from user
let p = 2
if p divides n, prime p
```

But that only checks for the case of `p` equals 2. You need to check the case when `p` equals 3, etc. That's repetition: so there must be a loop. The idea should look like this:

```
get n from user
let p = 2
while p does not divide n:
    p = p + 1
print p
```

Wait ... `p = p + 1` make `p` goes to the next integer, not the *next prime* after `p`. So it should look like this:

```
get n from user
let p = 2
while p does not divide n:
    p = next prime afer p
print p
```

The problem now is reduced to this idea of *next prime after*. To make the above pseudocode not overly messy, say we have a function:

```
int get_next_prime_after(int p):
    ... TO BE COMPLETE ...

int main():
    get n from user
    let p = 2
    while p does not divide n:
        p = get_next_prime_after(p)
    print p
```

(Remember that the above is pseudocode, not property C++.)

Clearly the get next prime function requires you to know how to check if a number is prime. So the above becomes:

```
int get_next_prime_after(int p):
    let i = p + 1
    while i is not prime:
        i = i + 1 (or ++i)

int main():
    get n from user
    let p = 2
    while p does not divide n:
        p = get_next_prime_after(p)
```

```
    print p
```

which means now you must have a function to check if a specific number is a prime:

```
bool is_prime(int i):
    ... TO BE COMPLETED ...

int get_next_prime_after(int p):
    let i = p + 1
    while !is_prime(i):
        i = i + 1 (or ++i)

int main():
    get n from user
    let p = 2
    while p does not divide n:
        p = get_next_prime_after(p)
    print p
```

The function call `is_prime(i)` returns `true` if the value of `i` is prime and it returns `false` if the value of `i` is not a prime. I will leave the logic of the function `bool is_prime(int)` to you since it's a pretty standard CISS240 type problem and in fact is in my C++ notes.

Note that on completing the above, you have a problem that computes (and prints) the first prime dividing `n`. After that you need to compute the power of that prime dividing `n`. How would you do that? Once you've found the smallest prime, say p, divding n, you just continually divide n by p until n is *not* divisible by p. For instance if $n = 1000$ and you already know that $p = 2$ divides n, then you would do this sequence of division:

$$n = 1000, \text{let } n = n/p, \text{then } n = 500$$
$$n = 500, \text{let } n = n/p, \text{then } n = 250$$
$$n = 250, \text{let } n = n/p, \text{then } n = 125$$

at which point n is not divisible by $p = 2$ anymore. Right? That's no mystery here. This is exactly what you did in elementary school.

Of course in the above you still need to keep a count of the power of 2 dividing 1000. All you need is a counter variable. So now you add the part that computes the prime power when a dividing prime is found:

```
bool is_prime(int i):
    ... TO BE COMPLETE ...
```

```
int get_next_prime_after(int p):
    let i = p + 1
    while !is_prime(i):
        i = i + 1 (or ++i)

int main():
    get n from user
    let p = 2
    while p does not divide n:
        p = get_next_prime_after(p)
    print p

    count = 0
    while n is divisible by p:
        count = count + 1
        n = n / p
    print count
```

This prints the *first* prime power dividing your n. What about the second? You just repeat the above process in a loop. After all if the input is $n = 1000$, after you've printed out the 2^3, your n becomes 125 and you can execute the same logic on $n = 125$ to get 5^3. Right?

There are other details, but the rest are minor. For instance you have to print the `*` and also, note that if the power is 1, you do not print it. (For the printing of `*`, note that if you have 4 prime powers, you only print 3 `*`'s.)

When you step back, you see that the highest level idea is something like this:

```
get n from user

while n is not 1:
    set p to the the smallest prime dividing n
    find k such that k is the largest p^k divides n
    set n to n / p^k
    print p and k
```

except that if you have already divided n by the maximal power of 2, there's no need to check for 2 again. So we keep a variable `p` to remember where we stopped:

```
get n from user

p = 1
while n is not 1:
    set p to the next prime (after p) dividing n
```

```
find k such that k is the largest p^k divides n
set n to n / p^k
print p and k
```

The next two questions involve the computation of magic squares. First, google and read up about magic squares.

For us an n-by-n magic square is an n-by-n grid filled with the numbers $1, 2, 3, ..., n^2$ such that each row and each column and each of the two diagonals add up to the same number. Here's a 3-by-3 magic square:

2	7	6
9	5	1
4	3	8

Our goal is to find magic squares.

One way (a very bad way) to find magic squares say for a 3-by-3 grid is to run through all the numbers 1–9 for each cell in the grid. Here's such a list of potential magic squares (including 0s as well):

0	0	0
0	0	0
0	0	0

0	0	0
0	0	0
0	0	1

0	0	0
0	0	0
0	0	2

...

9	9	9
9	9	9
9	9	7

9	9	9
9	9	9
9	9	8

9	9	9
9	9	9
9	9	9

Note that this is the same as running over the following sequence of arrays

```
0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,1
0,0,0,0,0,0,0,0,2
...
9,9,9,9,9,9,9,9,7
9,9,9,9,9,9,9,9,8
9,9,9,9,9,9,9,9,9
```

For each array, you simply check that if conditions for being a magic square are satisfied. If so, you have found a 3-by-3 magic square.

The above is an example of solving a problem by searching. This method is obviously not very smart therefore is not fast. In CS there are (at least) three general methods to search for solutions: brute force search, branch-and-bound search, and heuristic search. The above is a brute force search. In CISS350, we might have some search examples but we will not cover heuristic search – that's covered in the AI class. For another example of brute force search, look for the perfect number problem in the review problems set.

To use the above (bad) technique, we will need to generate the list

```
0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,1
0,0,0,0,0,0,0,0,2
...
9,9,9,9,9,9,9,9,7
9,9,9,9,9,9,9,9,8
9,9,9,9,9,9,9,9,9
```

This is only for the case of 1-9 for each cell, i.e., for the 3-by-3 magic square case. To find 4-by-4 magic squares, we need to fill the cells with values from 1 to 16. etc. Q4 deals with this problem.

Q5 then uses Q4 to generate potential magic squares and prints out only those that are really magic squares.

The point of the next two questions is just to practice computations using arrays (and to also introduce a particular search problem in CS.)

Q4. If you run a for-loop on `i` running from 0 to 100 and print out the values, you see the following where I've separated the digits of `i` by commas and also I've padded the integer so that there are exactly 4 digits:

```
0,0,0
0,0,1
0,0,2
...
0,0,9
0,1,0
0,1,1
0,1,2
0,1,3
...
0,9,8
0,9,9
1,0,0
```

Note that once a column c reaches `9`, the next number will result in that column being reset to `0` and a carry to the next column, i.e., column $c + 1$. You can also go through the same process but instead of performing a carry after `9`, you can have a carry after `15`:

```
0,0,0
0,0,1
0,0,2
...
0,0,9
0,0,10
0,0,11
0,0,12
0,0,13
0,0,14
0,0,15
0,1,0
0,1,1
0,1,2
0,1,3
...
0,15,14
0,15,15
1,0,0
```

And of course you can have more than 3 columns, say 4 columns:

```
0,0,0,0
0,0,0,1
0,0,0,2
...
0,0,0,9
0,0,0,10
0,0,0,11
0,0,0,12
0,0,0,13
0,0,0,14
0,0,0,15
0,0,1,0
0,0,1,1
0,0,1,2
0,0,1,3
...
0,0,15,14
0,0,15,15
0,1,0,0
```

You are basically working with base 16 numbers, i.e., hexadecimals. In the above case, the base is said to be 16. In our usual "human" number system, we use base 10. Let's put the digits into an array. For instance `0,0,15,14` can obviously be placed in an array `digits` of size 4 so that

```
digits[0] = 14, digits[1] = 15, digits[2] = 0, digits[3] = 0
```

Write a function

```
void increment(int digits[], int len_digits, int base);
```

that modifies the array `digits` to give you the next one as in the above list. So for instance if `digits` is the above array

```
digits[0] = 14, digits[1] = 15, digits[2] = 0, digits[3] = 0
```

then after calling

```
increment(digits, 4, 16)
```

we have

```
digits[0] = 15, digits[1] = 15, digits[2] = 0, digits[3] = 0
```

and calling it one more time gives us

```
digits[0] = 0, digits[1] = 0, digits[2] = 1, digits[3] = 0
```

Here's the skeleton:

```
#include <iostream>

void increment(int digits[], int len_digits, int base)
{
}

void print(int digits[], int len_digits)
{
    for (int i = len_digits - 1; i >= 1; --i)
    {
        std::cout << digits[i] << ',';
    }
    std::cout << digits[0] << std::endl;
}

int main()
{
    int digits[4] = {0};
    for (int i = 0; i < 100; ++i)
    {
        print(digits, 4);
        increment(digits, 4, 16);
    }
    return 0;
}
```

Q5. (Do not use `std::vector`.)

The following gives you the format that you must follow:

```
+--+--+--+
| 2| 7| 6|
+--+--+--+
| 9| 5| 1|
+--+--+--+
| 4| 3| 8|
+--+--+--+
+--+--+--+
| 4| 9| 2|
+--+--+--+
| 3| 5| 7|
+--+--+--+
| 8| 1| 6|
+--+--+--+
```

(This is *not* the complete list for 3-by-3 case – it's just for me to show you the format of the printout.) Of course in the case of a 4-by-4 magic square you have to fill each cell with numbers from 1 to $4^2 = 16$.

Write a program that accepts a value for n and prints all the n–by–n magic squares using the above method according to the above display format. You will of course need Q4 so that you can loop through all the potential n–by–n magic squares.

(You need to think about when to stop the loop. And you should think a little about improving on the method to improve the performance of the program.)