CISS240: Introduction to Programming
Assignment 10

Name: _____

Write a simple payroll program. There are three employee types (0, 1, and 2.) The computation of their salary is as follows:

- Employee types 0 and 1 have base pay of \$1000 and \$960.50 respectively. Employee type 2 does not have a base pay.
- All employees are paid hourly wage. For employee type 0, the hourly rate is \$10.50 per hour for regular work hours. The rate is \$8.50 and \$6.10 for employee types 1 and 2 respectively. The number of regular hours per week is 40.
- Employee type 0 has an overtime hourly rate of \$12.00 per hour. This is the rate for work beyond 40 hours. The overtime rate for an employee of type 1 is \$11.55. The overtime hourly rate for a type 2 employee is \$7.00.

The program prompts the user for the employee type and the number of hours worked. It then prints the salary earned for that employee and also the total salary to be paid out to all employees. When the user enters -1 for either the employee type or the number of hours, the program terminates.

You MUST use constants where applicable. (There should be at least 10.)

Test 1.

```
Employee type: 0
Hours of work: 0
Salary: $1000.00
Total: $1000.00

Employee type: 1
Hours of work: 0
Salary: $960.50
Total: $1960.50

Employee type: 2
Hours of work: 0
Salary: $0.00
Total: $1960.50

Employee type: -1
```

Test 2.

```
Employee type: 0
Hours of work: 10
Salary: $1105.00
Total: $1105.00

Employee type: 0
Hours of work: 50
Salary: $1540.00
Total: $2645.00

Employee type: -1
```

Test 3.

```
Employee type: 1
Hours of work: 10
Salary: $1045.50
Total: $1045.50

Employee type: 1
Hours of work: 60
Salary: $1531.50
Total: $2577.00

Employee type: -1
```

Test 4.

```
Employee type: 2
Hours of work: 10
Salary: $61.00
Total: $61.00

Employee type: 2
Hours of work: 70
Salary: $454.00
Total: $515.00

Employee type: -1
```

Write a program that continually prompts the user for doubles, printing the number of terms entered, the running sum, the running average, the running **minimum** and the running **maximum**, until the user enters 0. Except for the printing of the number of terms entered, all other output must be in fixed point format and set to precision of 5. If no data is entered, the program prints "no data entered".

Test 1.

```
0
no data entered
```

Test 2.

```
1
1 1.00000 1.00000 1.00000 1.00000
1
2 2.00000 1.00000 1.00000 1.00000
1
3 3.00000 1.00000 1.00000 1.00000
1
4 4.00000 1.00000 1.00000 1.00000
0
```

Test 3.

```
1.1
1 1.10000 1.10000 1.10000 1.10000
2.2
2 3.30000 1.65000 1.10000 2.20000
3.3
3 6.60000 2.20000 1.10000 3.30000
4.4
4 11.00000 2.75000 1.10000 4.40000
5.5
5 16.50000 3.30000 1.10000 5.50000
0
```

Test 4.

```
1.1
1 1.10000 1.10000 1.10000 1.10000
-2.2
2 -1.10000 -0.55000 -2.20000 1.10000
3.3
3 2.20000 0.73333 -2.20000 3.30000
-4.4
4 -2.20000 -0.55000 -4.40000 3.30000
5.5
5 3.30000 0.66000 -4.40000 5.50000
-6.6
6 -3.30000 -0.55000 -6.60000 5.50000
0
```

Write a simple text–based "game" where you control your avatar in a 2D world with x coordinates between 0 and 4 (inclusive) and y coordinates between 0 and 4 (inclusive). Your initial position in the 2D world is $x = 0$, $y = 0$. You control your avatar by moving in the North, South, East, West direction. For instance, moving it North will increase your y coordinate by 1, moving it West will decrement the x coordinate by 1. However, you cannot move outside the 2D world, i.e., your x and y coordinates must be between 0 and 4 (inclusive). For instance, if you're at $x = 2$, $y = 0$, moving South will not change your position; if you're at $x = 2$, $y = 4$, moving North will not change your position either. You enter 0 to move North, 1 to move South, 2 to move East, and 3 to move West. You can also enter 99 to end the game.

A pot of gold is randomly placed in the world except that the position cannot be $x = 0$, $y = 0$ (i.e., the initial position of your avatar.) You must write a while–loop to compute a position for the pot of gold until the position is not $x = 0$, $y = 0$. The pseudocode is as follows:

```
gold_x = random integer between 0 and 4
gold_y = random integer between 0 and 4
while gold_x is 0 and gold_y is 0:
    gold_x = random integer between 0 and 4
    gold_y = random integer between 0 and 4
```

where `gold_x` and `gold_y` are the x and y coordinates of the pot of gold.

The game ends either when you enter 99 to end the game or when you move onto the same position as the pot of gold. When the game ends, the number of moves is printed.

Your `main` must look like the following:

```
#include ...
#include ...

int main()
{
    std::cout << "seed: ";
    int n = 0;
    std::cin >> n;
    srand(n);

    // Your code here

    return 0;
}
```

You must use

```
rand() % 5
```

to obtain a random integer between 0 and 4 (inclusive).

The test cases are not exhaustive. You should include your own test cases.

Warning: These test cases are only examples to illustrate the output format for you. Yours will probably look different because of randomness.

Test 1.

```
seed: 0
gold hunting game!!!

position of pot: x=3, y=1
your position: x=0, y=0
0-north, 1-south, 2-east, 3-west, 99-quit: 99

number of moves: 0
```

Test 2.

```
seed: 5
gold hunting game!!!

position of pot: x=0, y=2
your position: x=0, y=0
0-north, 1-south, 2-east, 3-west, 99-quit: 0

position of pot: x=0, y=2
your position: x=0, y=1
0-north, 1-south, 2-east, 3-west, 99-quit: 1

position of pot: x=0, y=2
your position: x=0, y=0
0-north, 1-south, 2-east, 3-west, 99-quit: 2

position of pot: x=0, y=2
your position: x=1, y=0
0-north, 1-south, 2-east, 3-west, 99-quit: 3

position of pot: x=0, y=2
```

```
your position: x=0, y=0
0-north, 1-south, 2-east, 3-west, 99-quit: 99

number of moves: 4
```

Test 3. (This tests that you cannot move South when $x = 0$.)

```
seed: 17
gold hunting game!!!

position of pot: x=0, y=2
your position: x=0, y=0
0-north, 1-south, 2-east, 3-west, 99-quit: 1

position of pot: x=0, y=2
your position: x=0, y=0
0-north, 1-south, 2-east, 3-west, 99-quit: 99

number of moves: 1
```

Test 4. (This tests that you cannot move North when $y = 4$.)

```
seed: 39
gold hunting game!!!

position of pot: x=1, y=0
your position: x=0, y=0
0-north, 1-south, 2-east, 3-west, 99-quit: 0

position of pot: x=1, y=0
your position: x=0, y=1
0-north, 1-south, 2-east, 3-west, 99-quit: 0

position of pot: x=1, y=0
your position: x=0, y=2
0-north, 1-south, 2-east, 3-west, 99-quit: 0

position of pot: x=1, y=0
your position: x=0, y=3
0-north, 1-south, 2-east, 3-west, 99-quit: 0
```

```
position of pot: x=1, y=0
your position: x=0, y=4
0-north, 1-south, 2-east, 3-west, 99-quit: 0

position of pot: x=1, y=0
your position: x=0, y=4
0-north, 1-south, 2-east, 3-west, 99-quit: 99

number of moves: 5
```

(You should also test that your avatar cannot move beyond the left and right boundary of the 2D world.)

Test 5.

```
seed: 9001
gold hunting game!!!

position of pot: x=1, y=2
your position: x=0, y=0
0-north, 1-south, 2-east, 3-west, 99-quit: 0

position of pot: x=1, y=2
your position: x=0, y=1
0-north, 1-south, 2-east, 3-west, 99-quit: 0

position of pot: x=1, y=2
your position: x=0, y=2
0-north, 1-south, 2-east, 3-west, 99-quit: 2

you got the gold!
number of moves: 3
```

A perfect number is a positive integer which is the sum of its divisors strictly less than itself. For instance, the positive divisors of 6 are exactly 1, 2, 3, 6. The positive divisors which are strictly less than 6 are 1, 2, 3. Therefore, the sum of the positive divisors which are strictly less than 6 is $1+2+3$ which is 6. Here are some examples:

n	divisors of n strictly less than n	sum of divisors strictly less than n
1		0
2	1	1
3	1	1
4	1,2	3
5	1	1
6	1,2,3	6
7	1	1
8	1,2,4	7
9	1,3	4
10	1,2,5	8
11	1	1
12	1,2,3,4,6	16

Note that the sum of divisors of n which are strictly less than n can be less than n, equal to n, or greater than n. The goal of this program is to prompt the user for a, b and for each integer n from a to b (inclusive), print the sum of divisors which are strictly less than n, and "perfect"if n is a perfect number, and print the total number of perfect numbers found.

Note that you must not manually enter known perfect numbers in your code. Your code must search for them. For instance, from the above you know that 6 is a perfect number. You code must not contain something similar to this pseudocode: "If n is 6, print that it is a perfect number without checking its divisors."

You will see from your tests that perfect numbers are very rare. Although the ancient greek mathematicians and philosophers were interested in perfect numbers, before Euler, they did not know of too many: only 4. That was quite an amazing feat without computers. It probably took them hundreds of years to find the 4–th perfect number. Yet, with the right programming skills, we can discover all the first 4 perfect numbers in a second.

Do you see something common among all the perfect numbers in the above test cases? With the help of your program, can you discover the 5–th perfect number? Despite the simplicity of the definition of perfect numbers, there are still problems about perfect numbers which are not solved. For instance, it is not known if there are odd perfect numbers. [You can find out more about perfect numbers on Wikipedia.]

Test 1.

```
1 12
perfect number(s): 6
number of perfect numbers found: 1
```

Test 2.

```
10 20
number of perfect numbers found: 0
```

Test 3.

```
1 30
perfect number(s): 6 28
number of perfect numbers found: 2
```

Test 4.

```
1 100
perfect number(s): 6 28
number of perfect numbers found: 2
```

Test 5.

```
1 1000
perfect number(s): 6 28 496
number of perfect numbers found: 3
```

Test 6.

```
1 10000
perfect number(s): 6 28 496 8128
number of perfect numbers found: 4
```

An integer is a palindrome if it stays the same when you write its digits in reverse order. For instance, the integer 13531 is a palindrome. Write a program that prompts the user for an integer and displays 1 if the integer is a palindrome; otherwise 0 is printed.

Test 1.

```
0
1
```

Test 2.

```
5
1
```

Test 3.

```
12321
1
```

Test 4.

```
123321
1
```

Test 5.

```
1232
0
```

Test 6.

```
112311
0
```

Test 7.

```
1212
0
```

Test 8.

```
1210
0
```

Uh-oh . . . another ASCII art program.

Test 1.

```
1
  *
 ***
*****
```

Test 2.

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

Test 3.

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

Test 4.

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

Spoiler Warning . . . Incoming Hints . . .

Look at the case of Test 4:

Test 4

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

There are two possible outermost for-loop:

Either

```
for i = 1, 2, ..., 12
    [some code to draw 1 line]
```

or

```
for i = 1, 2, 3, 4:
    [some code to draw 3 lines]
```

In general, you can design your pseudocode as either

```
get value for n from user
for i = 1, 2, ..., 3 * n:
    [some code to draw 1 line]
```

or

```
get value for n from user
for i = 1, 2, ..., n:
    [some code to draw 3 lines]
```

It doesn't really matter which for–loop you use. You can solve this program using either form. The solution I will give uses the second form. For each i, you print a

layer of triangles. For instance, in the case of $n = 4$, when $i = 3$, your program should print the third layer of triangles:

```
...
     *     *     *
    ***   ***   ***
   ***** ***** *****
...
```

(layers 1, 2 and 4 not shown.)

The pseudocode should look something like this:

```
get a value for n from user

// this is the number of spaces to the left before
// anything is printed on each line
num_spaces = ???

for i in 1, 2, 3, 4, ..., n:
    print num_spaces of ' '
    num_spaces--
    print first line of the i-th layer of triangles

    print num_spaces of ' '
    num_spaces--
    print second line of the i-th layer of triangles

    print num_spaces of ' '
    num_spaces--
    print third line of the i-th layer of triangles
```

[Think about this: how many triangles are there at layer `i`?]