1 - Beginner - Out of Order Orders

Problem Statement

Frieda's *First Come, First Served FroYo* prides itself on serving customers in the exact order that they place their orders. Unfortunately, frozen yogurt machines are constantly in need of repair, so sometimes a customer's order is delayed and they have to wait while others who placed their order later receive their FroYo first. Frieda especially doesn't like it when an earlier order is picked up after one placed later.

Each order is assigned a number that is a positive integer, where each order number is always bigger than the last order issued. For example, if a customer places an order and receives order number 42, the very next order placed may be number 45 (their order system is a little glitchy).

Frieda uses a tracking system to determine when customers receive their food. The tracking system records a list of integers representing order numbers, which are listed in the order that the customers received their FroYo. Write a program to determine how many customers received their order directly after a larger order number than their own.

Input

Your program should take the following input:

- Six order numbers as integers, each falling between 0 and 100

You may assume that these are the only six orders received and that all customers receive their food.

Output

Your program should output the number of customer who receive their food directly after a larger order number as a positive integer.

Example

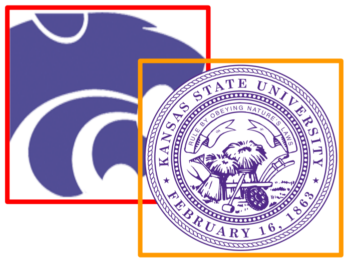
Input	**Output**
Order Numbers: **41 43 42 44 45 47**	Orders out of order: **1**
Order Numbers: **29 28 31 27 30 35 36**	Orders out of order: **2**

```python
"""
Model Solution
Beginner Problem 1: Out of Order
"""

# Read input
orders = [int(order) for order in input('Please enter the all 10 order numbers, separated by a space: ').split(' ')]
# Store initial order number
last = orders[0]
# Store number of orders out of order
out_of_order = 0
for order in orders:
    # If this order is less than the previous one
    if order < last:
        # Increase out of order count
        out_of_order += 1
    # Store this order as the new previous order
    last = order
print("Orders out of order: {}".format(out_of_order))

```

Problem Statement - B2

Box Collisions

Two dimensional (2D) video games often resolve multiple collisions every frame. If every collision was precise to the pixel, the game could quickly lose performance. One way to handle all these collisions is to use an Axis-Aligned Bounding Box (AABB). Every collidable object in a game utilizing AABBs is given a fixed (i.e. non-rotating) rectangle, and collision checks simply test if two rectangles intersect. This can be seen in the figure below.



Example of an AABB Intersection

KANSAS STATE UNIVERSITY
FEBRUARY 16, 1863

Example of no AABB Intersection

Write a program that implements AABBs as described above. This program should accept a constant width for all rectangles, a constant height for all rectangles, and the unique x/y coordinates for the bottom left corner of two different rectangles. The program should then print if there is a collision or not between the two rectangles. Note that a collision should only occur if the rectangles are overlapping and not just touching. You can assume that the values for width, height, x, and y are non-zero positive integers.

Example 1	Example 2
Input: Enter width for rectangles: 4 **Input:** Enter height for rectangles: 6 **Input:** Enter x for rectangle 1: 2 **Input:** Enter y for rectangle 1: 3 **Input:** Enter x for rectangle 2: 4 **Input:** Enter y for rectangle 2: 7 **Output:** There is a collision	**Input:** Enter width for rectangles: 2 **Input:** Enter height for rectangles: 2 **Input:** Enter x for rectangle 1: 4 **Input:** Enter y for rectangle 1: 2 **Input:** Enter x for rectangle 2: 6 **Input:** Enter y for rectangle 2: 2 **Output:** There is no collision

Solution - B2

Box Collisions

```
using System;

public class Program
{
    public static void Main(string[] args)
    {
        Console.Write("Enter Width: ");
        int w = Convert.ToInt32(Console.ReadLine());
        Console.Write("Enter Height: ");
        int h = Convert.ToInt32(Console.ReadLine());

        Console.Write("Enter Rectangle 1's X: ");
        int x1 = Convert.ToInt32(Console.ReadLine());
        Console.Write("Enter Rectangle 1's Y: ");
        int y1 = Convert.ToInt32(Console.ReadLine());

        Console.Write("Enter Rectangle 2's X: ");
        int x2 = Convert.ToInt32(Console.ReadLine());
        Console.Write("Enter Rectangle 2's Y: ");
        int y2 = Convert.ToInt32(Console.ReadLine());

        if( x1 + w > x2 &&
            x1 < x2 + w &&
            y1 + h > y2 &&
            y1 < y2 + h )
        {
            Console.WriteLine("There is a collision");
        }
        else
        {
            Console.WriteLine("There is no collision");
        }
    }
}
```

3 Beginning — Magic Squares
Problem Statement

A 3×3 *magic square* is nine integers arranged in a 3×3 grid such that each row, column, and diagonal sums to the same value. For example, in each of the following magic squares, each row, column, and diagonal sums to 15:

2	9	4
7	5	3
6	1	8

3	11	1
3	5	7
9	−1	7

Note that we make no requirement that the values be unique or restricted to a particular range.

Write a program that takes as input three integers for the top row, from left to right, and an integer for the leftmost element of the middle row, and displays a magic sqaure containing these values, if possible. If no magic square exists with the given values in the given locations, a message to this effect should be displayed.

Example 1:

```
Enter value at row 0, column 0: 10
Enter value at row 0, column 1: -2
Enter value at row 0, column 2: 1
Enter value at row 1, column 0: -6

10 -2 1
-6 3 12
5 8 -4
```

Example 2:

```
Enter value at row 0, column 0: 1
Enter value at row 0, column 1: 1
Enter value at row 0, column 2: 3
Enter value at row 1, column 0: 4

No magic square exists.
```

```
// 3 - Beginning Magic Squares
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Beginning.MagicSquares
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter value at row 0, column 0: ");
            int topLeft = Convert.ToInt32(Console.ReadLine());
            Console.Write("Enter value at row 0, column 1: ");
            int topCenter = Convert.ToInt32(Console.ReadLine());
            Console.Write("Enter value at row 0, column 2: ");
            int topRight = Convert.ToInt32(Console.ReadLine());
            Console.Write("Enter value at row 1, column 0: ");
            int centerLeft = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine();
            int sum = topLeft + topCenter + topRight;             // Row 0
            int bottomLeft = sum - topLeft - centerLeft;          // Column 0
            int center = sum - bottomLeft - topRight;             // Minor diagonal
            int centerRight = sum - centerLeft - center;          // Row 1
            int bottomCenter = sum - topCenter - center;          // Column 1
            int bottomRight = sum - topRight - centerRight;       // Column 2
            if (bottomLeft + bottomCenter + bottomRight == sum && // Row 2 (unnecessary)
                topLeft + center + bottomRight == sum)            // Major diagonal
            {
                Console.WriteLine(topLeft + " " + topCenter + " " + topRight);
                Console.WriteLine(centerLeft + " " + center + " " + centerRight);
                Console.WriteLine(bottomLeft + " " + bottomCenter + " " + bottomRight);
            }
            else
            {
                Console.WriteLine("No magic square exists.");
            }
            Console.ReadLine();
        }
    }
}

```

4 Beginning — Card Sequence
Problem Statement

A deck of cards has been arranged into a particular sequence. The deck contains 32 cards, each of which is uniquely identified by a *rank* and a *suit*. The eight ranks are 7, 8, 9, 10, Jack, Queen, King, and Ace. The four suits are Hearts, Diamonds, Clubs, and Spades. Hearts and Diamonds are colored red, whereas Clubs and Spades are colored black.

The deck is arranged so that, no matter how it is cut by moving cards from the top to the bottom, the colors of the first five cards uniquely determine the first card. The colors of the first two cards determine the suit as follows:

- Black-Black: Spades
- Black-Red: Clubs
- Red-Black: Diamonds
- Red-Red: Hearts

The colors of the third, fourth, and fifth cards determine the rank as follows:

- The rank is initialized to 7.
- If the third card is black, add 4.
- If the fourth card is black, add 2.
- If the fifth card is black, add 1.

If the rank, as computed above, is 11, 12, 13, or 14, it is interpreted as Jack, Queen, King, or Ace, respectively.

Write a program that takes as input a sequence of five letters, each of which is either "B" (denoting black) or "R" (denoting red), and produces as output the rank and suit of the card these colors identifiy. You may choose whether to have the program accept the input as five separate letters or as a single string. You may assume that each character is either "B" or "R". Your output must be the full name of the card, as shown in the examples below.

Example 1:

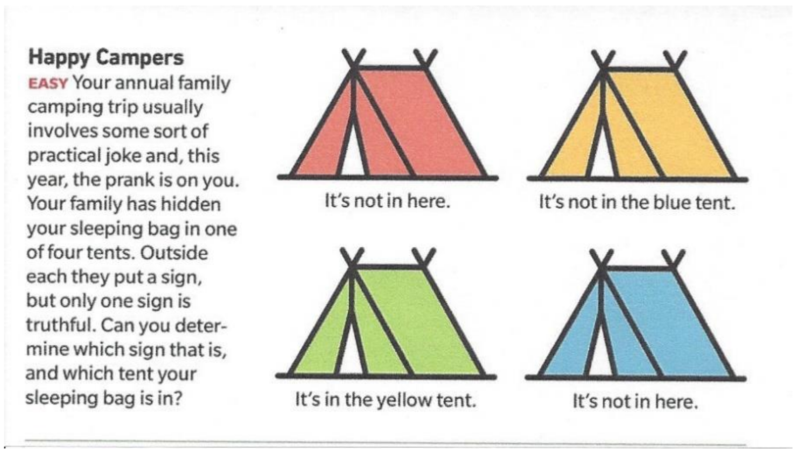
```
Enter first card's color: B
Enter second card's color: R
Enter third card's color: B
Enter fourth card's color: R
Enter fifth card's color: B

Queen of Clubs
```

Example 2:

```
Enter first card's color: R
Enter second card's color: R
Enter third card's color: R
Enter fourth card's color: B
Enter fifth card's color: B

10 of Hearts
```

```csharp
// 4 - Beginning Card Sequence
// See also the solution to the advanced problem for this round.
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Beginning.CardSequence
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter first card's color: ");
            string col1 = Console.ReadLine();
            Console.Write("Enter second card's color: ");
            string col2 = Console.ReadLine();
            Console.Write("Enter third card's color: ");
            string col3 = Console.ReadLine();
            Console.Write("Enter fourth card's color: ");
            int r = 0;
            if (Console.ReadLine() == "B") r += 2;
            Console.Write("Enter fifth card's color: ");
            if (Console.ReadLine() == "B") r += 1;
            string rank;
            if (col3 == "B")
            {
                if (r == 0) rank = "Jack";
                else if (r == 1) rank = "Queen";
                else if (r == 2) rank = "King";
                else rank = "Ace";
            }
            else rank = (r + 7).ToString();
            Console.WriteLine();
            if (col1 == "B")
            {
                if (col2 == "B") Console.WriteLine(rank + " of Spades");
                else Console.WriteLine(rank + " of Clubs");
            }
            else
            {
                if (col2 == "B") Console.WriteLine(rank + " of Diamonds");
                else Console.WriteLine(rank + " of Hearts");
            }
            Console.ReadLine();
        }
    }
}
```

B5 Problem Statement



Happy Campers

EASY Your annual family camping trip usually involves some sort of practical joke and, this year, the prank is on you. Your family has hidden your sleeping bag in one of four tents. Outside each they put a sign, but only one sign is truthful. Can you determine which sign that is, and which tent your sleeping bag is in?

It's not in here.
It's not in the blue tent.
It's in the yellow tent.
It's not in here.

There are many ways to solve logic problems like the one on the left. One approach is to consider the four cases: which signs are true if the sleeping bag is (1) in the red tent, (2) in the yellow tent, (3) in the blue tent, finally, (4) in the green tent. In the example on the left, if the sleeping bag is in the red tent: then the sign for the red tent is false, the sign for the yellow tent is true, the sign for the blue tent is true, and the sign for the green tent is false. In this case, there are two signs that are true and thus the sleeping bag cannot be in the red tent.

Solve the happy campers' problem for three tents: red, yellow, blue, and limited kinds of signs: "it's in <color> tent" or "it's not in <color> tent".

The program must accept the sign for each tent: 'T' mean the statement is positive. 'N' means it is negative. The letters R, Y, B mean red, yellow, and blue respectively.

Print out which tent contains the sleeping bag and which tent's sign is true.

If there is no situation in which only one sign is true, print out: no solution. If more than one case has only one sign true, print out either case.

Example 1:

Input: N R N B T R (means the red tent's sign is "it's not in the red tent"; the yellow tent's sign is "it's not in the blue"; and the blue test's sign is "it's in the red tent".)

Output: Bag in **blue** tent

True sign on **red tent**

Example 2:

Input: N R N Y N B

Output: **no solutio**n (all cases have two signs true)

```cpp
// LogicB.cpp : This file contains the 'main' function. Program execution begins
   and ends there.
//

#include <iostream>

int main()
{
    char TR, TY, TB, CR, CY, CB;
    char PR, PY, PB;
    int IsTR, IsTY, IsTB;
    char Quit;
    Quit = 'Q';
    while (Quit == 'Q')
    {
        std::cout << "\nEnter T or N for red";
        std::cin >> TR;
        std::cout << "\nEnter color for red";
        std::cin >> CR;

        std::cout << "\nEnter T or N for yellow";
        std::cin >> TY;
        std::cout << "\nEnter color for yellow";
        std::cin >> CY;

        std::cout << "\nEnter T or N for blue";
        std::cin >> TB;
        std::cout << "\nEnter color for blue";
        std::cin >> CB;


        IsTR = 0; IsTY = 0;  IsTB = 0;
        PR = 'X'; PY = 'V'; PB = 'W';

        std::cout << "\n Case: " << TR << CR << TY << CY << TB << CB << "\n1";

        // case 1: bag is in red tent

        if (TR == 'T' && CR == 'R') { IsTR++; PR = 'R'; }
        if (TR == 'N' && (CR == 'B' || CR == 'Y' )) { IsTR++; PR = 'R'; }
        if (TY == 'T' && CY == 'R') { IsTR++; PR = 'Y'; }
        if (TB == 'T' && CB == 'R') { IsTR++; PR = 'B'; }
        if (TY == 'N' && ( CY == 'B' || CY == 'Y')) { IsTR++; PR = 'Y'; }
        if (TB == 'N' && ( CB == 'B' || CB == 'Y')) { IsTR++; PR = 'B'; }

        // case 2: bag is in yellow tent

        if (TY == 'T' && CY == 'Y') { IsTY++; PY = 'Y'; }
        if (TY == 'N' && (CY == 'B' || CY == 'R')) { IsTY++; PY = 'Y'; }
        if (TR == 'T' && CR == 'Y') { IsTY++; PY = 'R'; }
        if (TB == 'T' && CB == 'Y') { IsTY++; PY = 'B'; }
        if (TR == 'N' && ( CR == 'B' || CR == 'R')) { IsTY++; PY = 'R'; }
        if (TB == 'N' && ( CB == 'B' || CB == 'R')) { IsTY++; PY = 'B'; }

        // case 3: bag is in blue tent

        if (TB == 'T' && CB == 'B') { IsTB++; PB = 'B'; }
        if (TB == 'N' && (CB == 'R' || CB == 'Y')) { IsTB++; PB = 'B'; }
        if (TY == 'T' && CY == 'B') { IsTB++; PB = 'Y'; }
        if (TR == 'T' && CR == 'B') { IsTB++; PB = 'R'; }
        if (TY == 'N' && (RCY == 'Y' || CY == 'R')) { IsTB++; PB = 'Y'; }
        if (TR == 'N' && (CR == 'Y' || CR == 'R')) { IsTB++; PB = 'R'; }

        std::cout << "\nR " << IsTR << " PR " << PR;
        std::cout << "\nY " << IsTY << " PY " << PY;
        std::cout << "\nB " << IsTB << " PB " << PB;

        if (IsTR == 1) std::cout << "\nbag is in Red, sign is at tent " << PR;
        if (IsTY == 1) std::cout << "\nbag is in Yellow, sign is at tent " << 
          PY;
        if (IsTB == 1) std::cout << "\nbag is in Blue,  sign is at tent " << PB;

        std::cout << "\nEnter quit";
        std::cin >> Quit;
    }
}


```

6 - Beginner - Super Secret

Problem Statement

Substitution ciphers are simple encryption algorithms where plaintext is replaced with ciphertext based off some fixed system. For example, we could shift or rotate the alphabet by some number of positions as the basis for encrypting a message. Rotating the alphabet by 13 places is known as a ROT13 cipher. In ROT13, a letter is substituted with a letter that comes 13 places after it. For example, if we start with a normal English alphabet, letters 'a' through 'm' will be substituted with a letter that comes 13 letters **after** the letter and letters 'n' through 'z' will be substituted with a letter that comes 13 **before** the letter. Note that this works in both directions (i.e. encryption and decryption are the same operation)

Write a program that will encrypt and decrypt messages using the ROT13 cipher. The program should take as input a message to encode or decode. The message should be encrypted/decrypted, and the result should be printed out.

Input

- A non-empty string that is the message to encode or decode. This message will only contain lower case alpha characters.

Output

Your program should output the message from the user that is encrypted or decrypted based off of the inputs.

Example

Input	Output
hello	**uryyb**
cookie	**pbbxvr**
tbpngf	**gocats**

```python
"""
Beginner problem 6: Super Secret

Note that this can be solved in many ways, including
character addition/subtraction, list manipulation, string manipulation,
or by hard-coding the substitutions using if statements.
"""
# alphabet for the regular plain text
plain_alphabet = "abcdefghijklmnopqrstuvwxyz"

message = input("Enter a message: ")

result = ""
for c in message:
    # get the index of the letter in the alphabet
    letter_index = plain_alphabet.index(c)

    # add or subtract 13 depending on which side of the
    #       alphabet it is in
    if letter_index < 13:
        result += plain_alphabet[letter_index+13]
    else:
        result += plain_alphabet[letter_index-13]
print(result)

```