Write a program that reads 4 positive integers:

- the numerator of \( x \);
- the denominator of \( x \);
- the numerator of \( y \); and
- the denominator of \( y \);

and displays a message indicating how \( x \) and \( y \) compare (i.e., whether \( x < y \), \( x = y \), or \( x > y \)). You may assume that all inputs are positive and less than 10,000. (Note: Although a solution using floating-point arithmetic might pass all the tests, a safer solution is to find a way to use only integers.)

Example 1:

Enter numerator of \( x \): 5
Enter denominator of \( x \): 2
Enter numerator of \( y \): 8
Enter denominator of \( y \): 3

\( x < y \)

Example 2:

Enter numerator of \( x \): 4
Enter denominator of \( x \): 6
Enter numerator of \( y \): 6
Enter denominator of \( y \): 9

\( x = y \)
namespace Fractions.Begining {

class Program {
    static void Main(string[] args) {
        Console.Write("Enter numerator of x: ");
        int xNum = Convert.ToInt32(Console.ReadLine());
        Console.Write("Enter denominator of x: ");
        int xDenom = Convert.ToInt32(Console.ReadLine());
        Console.Write("Enter numerator of y: ");
        int yNum = Convert.ToInt32(Console.ReadLine());
        Console.Write("Enter denominator of y: ");
        int yDenom = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine();
        // By multiplying both fractions by the product of the denominators, we obtain xNum * yDenom and yNum * xDenom
        if (xNum * yDenom < yNum * xDenom) {
            Console.WriteLine("x < y");
        } else if (xNum * yDenom == yNum * xDenom) {
            Console.WriteLine("x = y");
        } else {
            Console.WriteLine("x > y");
        }
        Console.ReadLine();
    }
}
*/
Test Case 1:

Enter numerator of x: 8888
Enter denominator of x: 3333
Enter numerator of y: 5555
Enter denominator of y: 2222

\[ x > y \]

Test Case 2:

Enter numerator of x: 2468
Enter denominator of x: 3702
Enter numerator of y: 2002
Enter denominator of y: 3003

\[ x = y \]

Test Case 3:

Enter numerator of x: 2468
Enter denominator of x: 3702
Enter numerator of y: 3333
Enter denominator of y: 4444

\[ x < y \]

**Second Submission:** Do the above tests, plus the following:

Test Case 4:

Enter numerator of x: 9999
Enter denominator of x: 7777
Enter numerator of y: 9
Enter denominator of y: 7

\[ x = y \]

Test Case 5:

Enter numerator of x: 9999
Enter denominator of x: 9998
Enter numerator of y: 9998
Enter denominator of y: 9997

\[ x < y \]
A boat floats because it displaces water. The weight of the water displaced by a boat is equal to the weight of the boat and the passengers.

Your task is to determine the minimum length of a rectangular boat given the total weight of the occupants. Assume the width of the boat is 2 feet, the depth of the boat is 2 feet and one foot of the depth is above the water line. Also assume that the weight of the boat is 100 lbs and is independent of the length.

Prompt for the total weight in pounds of the prospective occupants and output the length in feet.

The weight of one cubic foot of water at 70 degrees Fahrenheit is 62.3 lbs.

Example 1:

Total occupant weight: 200  
Output: 2.4

Example 2:

Total occupant weight: 400  
Output: 4.0
2B float your boat        solution

// weight of 1 cubic ft of water is 62.30 lbs/ft³ at 70 degrees Fahrenheit
#define WaterWeight 62.30
#define Height 1.0
#define Width 2.0

int _tmain(int argc, _TCHAR* argv[])
{
    double weight, length, volume;
    cout<<"Enter total weight of passengers: ";
    cin>>weight;
    // weight of boat needs to be added to occupant weight
    weight = weight + 100;
    // volume of water that needs to be displaced
    volume = weight/WaterWeight;
    // length*width*height = volume displaced
    length = volume/(Width*Height);
    cout<<"Length is "<<length;

    return 0;
}
2B float your boat Test Cases

**First testing:** Do the test cases listed below. The rounding off to the first decimal place is fine. Formatting is not important.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Total occupants weight: 100</td>
<td>length: 1.6</td>
</tr>
<tr>
<td>2</td>
<td>Total occupants weight: 321</td>
<td>length: 3.37</td>
</tr>
<tr>
<td>3</td>
<td>Total occupants weight: 500</td>
<td>length: 4.81</td>
</tr>
</tbody>
</table>

**Second testing:** Repeat the test cases above and do the additional ones below

<table>
<thead>
<tr>
<th>Test case</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Total occupants weight: 150</td>
<td>length: 2.0</td>
</tr>
<tr>
<td>5</td>
<td>Total occupants weight: 400</td>
<td>length: 4.0</td>
</tr>
</tbody>
</table>
We can assign all of the ordered pairs of natural numbers a unique natural number by placing the natural numbers on the $x$-$y$ coordinate system as follows. We first place 0 at location $(0, 0)$. We then place 1 above this location at $(0, 1)$, and continue around $(0, 0)$ clockwise, placing 2 at $(1, 1)$, 3 at $(1, 0)$, etc., until we place 8 at $(-1, 1)$. We then go to $(0, 2)$ and place 9, and continue spiraling around like this. A portion of the coordinate system looks like this:

```
46 47 48 25 26 27 28
45 23 24 9 10 11 29
44 22 8 1 2 12 30
43 21 7 0 3 13 31
42 20 6 5 4 14 32
41 19 18 17 16 15 33
40 39 38 37 36 35 34
```

Write a program that takes a nonnegative integer $y$ as input and displays the number at location $(0, y)$ in this spiral. You may assume that $y$ is no more than 1,000.

**Example 1:**

```
Enter y: 0
Location (0, 0) contains 0
```

**Example 2:**

```
Enter y: 4
Location (0, 4) contains 49
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Number.Spiral.Beginning
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter y: ");
            int y = Convert.ToInt32(Console.ReadLine());
            if (y == 0)
            {
                Console.WriteLine("Location (0, 0) contains 0");
            }
            else
            {
                int sum = 1;
                for (int i = 1; i < y; i++)
                {
                    sum += 8 * i;
                }
                Console.WriteLine("Location (0, " + y + ") contains "+ sum);
                // Or to avoid the loop, simply display (2*y-1)*(2*y-1)
            }
            Console.ReadLine();
        }
    }
}
Test Case 1:

Enter y: 0
Location (0, 0) contains 0

Test Case 2:

Enter y: 1
Location (0, 1) contains 1

Test Case 3:

Enter y: 1000
Location (0, 1000) contains 3996001

Second Submission: Do the above tests, plus the following:

Test Case 4:

Enter y: 307
Location (0, 307) contains 375769

Test Case 5:

Enter y: 747
Location (0, 747) contains 2229049
This problem is to simulate raising rabbits. The inputs are the initial number of breeding pairs and the length of time in months.

Rabbits can produce 4-12 baby rabbits every 30 days. Assume an average of 6 baby rabbits, half female. Rabbits mature to breed about seven months after birth. That is, the rabbits born at the end of month 1 will have babies at the end of month 8.

Assume no deaths during the specified time.

Print the total number of adults and juveniles at the end of the simulation.

Examples:

1) Initial pairs: 3  length: 14  Output adults: 132  juveniles: 1260
2) Initial pairs: 1  length: 6  Output adults:  2  juveniles:  36
3) Initial pairs: 2  length: 12  Output adults:  64  juveniles:  444
float length;
int babies[7];
int pairs, sum;
int newbabies;
int i, j;

cout<<"Enter number of breeding pairs: ";
cin>>pairs;
cout<<"Enter number of months: ";
cin>>length;
for(j=0; j<7; j++) babies[j]=0;

for (i=0; i<length; i++) {
    newbabies = 3*pairs;
pairs = pairs + babies[6];

    sum = 0;
    for (j=6; j>0; j--) {babies[j] = babies[j-1]; sum+=sum+babies[j];}
    babies[0] = newbabies; sum+=sum+babies[0];
cout<< "After " << i+1 << " months: " << 2*pairs << " adults " <<
<<2*sum << " babies " << newbabies*2 << " newbabies";
}
Rabbits

Test Cases

**First testing:** Do the test cases listed below. The results are all integer. Formatting is not important.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>initial pairs: 1</td>
<td>adults: 2</td>
</tr>
<tr>
<td></td>
<td>length: 6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>initial pairs: 3</td>
<td>adults: 132</td>
</tr>
<tr>
<td></td>
<td>length: 14</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>initial pairs: 5</td>
<td>adults: 100</td>
</tr>
<tr>
<td></td>
<td>length: 10</td>
<td></td>
</tr>
</tbody>
</table>

**Second testing:** Repeat the test cases above and do the additional ones below

<table>
<thead>
<tr>
<th>Test case</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>initial pairs: 2</td>
<td>adults: 232</td>
</tr>
<tr>
<td></td>
<td>length: 17</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>initial pairs: 3</td>
<td>adults: 42</td>
</tr>
<tr>
<td></td>
<td>length: 9</td>
<td></td>
</tr>
</tbody>
</table>
We are driving down the street and see a green traffic light ahead. Because we know precisely the pattern of this traffic light, we know exactly how long we have before it will turn red. We wish to compute whether we will pass the traffic light before it turns red at our current speed. Write a program that takes the following positive floating-point numbers as input:

- our current speed in miles per hour;
- the distance to the light in miles; and
- the time until it turns red in seconds;

and displays a message indicating whether we will beat the light. You may assume that the input won’t be such that we reach the light at exactly the time it turns red.

Example 1:

Enter speed: 59.99  
Enter distance to light: 1.0  
Enter time to red: 60.0

We won’t beat the light.

Example 2:

Enter speed: 60.01  
Enter distance to light: 1  
Enter time to red: 60

We will beat the light.
namespace Green.Light.Beginning
{
    public class Program
    {
        static void Main(string[] args)
        {
            double speed = Convert.ToDouble(Console.ReadLine());
            double distance = Convert.ToDouble(Console.ReadLine());
            double time = Convert.ToDouble(Console.ReadLine()) / 3600.0;
            Console.WriteLine();
            if (speed > distance / time)
            {
                Console.WriteLine("We will beat the light.");
            }
            else
            {
                Console.WriteLine("We won't beat the light.");
            }
            Console.ReadLine();
        }
    }
}
Test Case 1:

Enter speed: 30  
Enter distance to light: 0.25  
Enter time to red: 29.9

We won’t beat the light.

Test Case 2:

Enter speed: 40.1  
Enter distance to light: 0.1  
Enter time to red: 9

We will beat the light.

Test Case 3:

Enter speed: 45.6  
Enter distance to light: 0.12  
Enter time to red: 7.6

We won’t beat the light.

Second Submission: Do the above tests, plus the following:

Test Case 4:

Enter speed: 62.3  
Enter distance to light: 0.001  
Enter time to red: .001

We won’t beat the light.

Enter speed: 15.2  
Enter distance to light: .05  
Enter time to red: 13.7

We will beat the light.
This problem is to calculate the minimum number of coins needed to make specific change. Assume that there are half dollars, quarters, dimes, nickels and pennies available. Input is the total bought in dollars and the amount paid. The output should be the number of each coin needed.

Use a greedy algorithm that first chooses as many as possible of the largest coin, then uses the same strategy for the next largest coin, etc.

Examples:

1) bought: 2.87  paid:  3.00  Output: 0 half, 0 quarter, 1 dime, 0 nickel, 3 penny
2) bought: 1.67  paid:  3.00  Output: 2 half, 1 quarter, 0  dime, 1 nickel, 3 penny
3) bought: .89   paid:  1.00  Output: 0 half, 0 quarter, 1  dime, 0 nickel, 1 penny
4) bought: 2.30  paid:  5.00  Output: 4 half, 2 quarter, 2  dime, 0 nickel, 0 penny
int dollar, half, quarter, dime, nickel, penny;
float purchase, tendered, change;

char q; q = 'q';
while (q == 'q') {
    cout<<"enter cost of purchase: ";
    cin>>purchase;
    cout<<"enter amount tendered: ";
    cin>>tendered;

    change = tendered - purchase;
    cout<<"\n amount to be returned: ",change;
    //convert dollars and cents to pennies
    change = change*100;
    cout<<"\n change ",
    dollar = 0;
    while ( change >= 100) {dollar++; change = change - 100;}
    cout<<"\n dollars needed: ",dollar;

    half = 0;
    while ( change >= 50) {half++; change = change - 50;}
    cout<<"\n 50 cents needed: ",half;

    quarter = 0;
    while ( change >= 25) {quarter++; change = change - 25;}
    cout<<"\n quarters needed: ",quarter;

    dime = 0;
    while ( change >= 10) {dime++; change = change - 10;}
    cout<<"\n dimes needed: ",dime;

    nickel = 0;
    while ( change >= 5) {nickel++; change = change - 5;}
    cout<<"\n nickels needed: ",nickel;

    penny = 0;
    while ( change >= 1) {penny++; change = change - 1;}
    cout<<"\n pennies needed: ",penny;
First testing: Do the test cases listed below. The output should be integers. Formatting is not important.

Test case

1) bought: **2.27** paid: **5.00**

Output: **7** half, **0** quarters, **2** dimes, **0** nickels, **3** penny

2) bought: **3.58** paid: **5.00**

Output: **2** half, **1** quarters, **1** dimes, **1** nickels, **2** penny

3) bought: **0.27** paid: **1.00**

Output: **1** half, **0** quarters, **2** dimes, **0** nickels, **3** penny

Second testing: Repeat the test cases above and do the additional ones below

Test case

4) bought: **2.09** paid: **3.00**

Output: **1** half, **1** quarter, **1** dime, **1** nickel, **1** penny

5) bought: **4.93** paid: **5.00**

Output: **0** half, **0** quarter, **0** dime, **1** nickel, **2** penny