Bloomsburg University

20th Annual High School Programming Contest

2015

Department of Mathematics, Computer Science, and Statistics

1. Buy One, Get One Free

I went into business selling homemade apple pies from my garage. Here are the prices:

Small pie:	$2
Medium pie:	$3
Large pie:	$5

Unfortunately, due to health code violations, I am going out of business. So I am having a big sale and here's the deal: customers can buy any pie and get a second pie of the same size for free. However, if a customer buys three pies of the same size, then the full price will be charged for all three. A customer may purchase at most three pies.

Write a program that prompts the user to enter a customer's purchase and then outputs the total price to be paid.

A purchase is specified by a sequence of letters (S, M, and/or L) indicating the size of each pie. For example, "MSM" denotes a purchase of two medium pies and one small pie. In this case, the output would be $5 since one of the medium pies would be free.

The order of letters in a purchase does not matter (so, for example, MSM is equivalent to both SMM and MMS).

The required I/O format is illustrated by the following execution snapshots:

```
Enter purchase: MSM
Price: $5
```

```
Enter purchase: SML
Price: $10
```

```
Enter purchase: S
Price: $2
```

```
Enter purchase: MM
Price: $3
```

```
Enter purchase: MMM
Price: $9
```

2. Pacer

One way that distance runners increase their speed is by running intervals at the track. This means to repeatedly run a certain fixed distance (an interval) at a precise speed with slow jogs in between. Tracks are marked with intervals measured in meters, but distance runners often think in terms of their mile pace (the time it would take to run a mile at that speed).

For example, a runner might want to run 400-meter intervals at a 5:15 mile pace. There are 1609 meters in a mile, so the runner would need to run each interval in 1:18.

Write a program that allows the user to enter the interval distance in meters and the target mile pace, and then outputs the time required to run each interval. The interval distance will be a whole number not more than 1600 meters. Fractions of a second should be truncated.

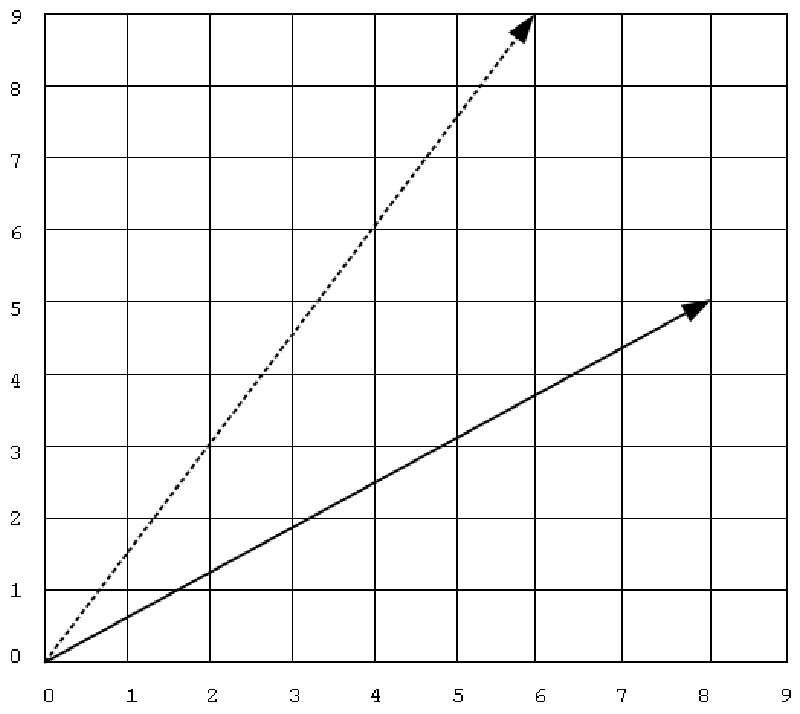
The required I/O format is illustrated by the following execution snapshots:

```
Interval distance in meters: 400
Target mile pace: 5:15
Time for each interval: 1:18

Interval distance in meters: 800
Target mile pace: 6:45
Time for each interval: 3:21

Interval distance in meters: 200
Target mile pace: 7:05
Time for each interval: 0:52

Interval distance in meters: 600
Target mile pace: 8:25
Time for each interval: 3:08
```

Times entered by the user and written by the program must be in M:SS format, as shown above (7:05 represents 7 minutes and 5 seconds, for example, and 0:52 represents 52 seconds).

3. Line of Sight

Let L denote the set of all points in the plane with integer coordinates. Given a positive integer n, we will write $L(n)$ for those points (x, y) in L with $0 \le x \le n$ and $0 \le y \le n$. For this problem, we will be interested in points of $L(n)$ that are "visible from the origin" in the sense that you can draw a straight line to them from the origin without passing through any intervening points. For example, if $n = 9$, we have this picture in mind:



You can see that the point at (8, 5) is visible from the origin. But the point (6, 9) is *not* visible from the origin – its view is blocked by the points at (2, 3) and (4, 6).

Write a program that prompts the user for a positive integer n and then outputs the number of points in $L(n)$ that are visible from the origin.

Execution snapshots:

```
Enter a positive integer: 3
Points of L(3) visible from the origin: 9

Enter a positive integer: 10
Points of L(10) visible from the origin: 65

Enter a positive integer: 42
Points of L(42) visible from the origin: 1085

Enter a positive integer: 999
Points of L(999) visible from the origin: 607585
```

4. SLATIPAC

Write a program that takes as input a line of text (letters, digits, punctuation) and then outputs the string obtained from the input by reversing all substrings consisting entirely of capital letters.

Execution snapshots:

```
 Input: GABCFabc
Output: FCBAGabc

 Input: 123abcAZBCDExyzXSTZ
Output: 123abcEDCBZAxyzZTSX

 Input: abcdefAABxyz
Output: abcdefBAAxyz

 Input: AbCCDD-2EfghPONY
Output: AbDDCC-2EfghYNOP
```

5. Marbles

In a bowl of *n* marbles colored black or white, the probability of drawing two black marbles in a row is:

$$\frac{k}{n} \times \frac{k-1}{n-1}$$

where *k* is the number of black marbles in the bowl.

Write a program that prompts the user for a fraction $f = a/b$ and outputs the number of black and white marbles in the smallest collection for which the probability of selecting two black marbles in a row is exactly *f*.

The user will specify a fraction by entering its numerator and denominator separated by a space.

Sample execution (with user input in **bold**):

```
Numerator and denominator: 2 5
4 black and 2 white
```

Note that with 4 black and 2 white marbles, the probability of drawing two black marbles in a row is

$$\frac{4}{6} \times \frac{3}{5} = \frac{2}{5}.$$

This is the smallest collection of marbles having that property.

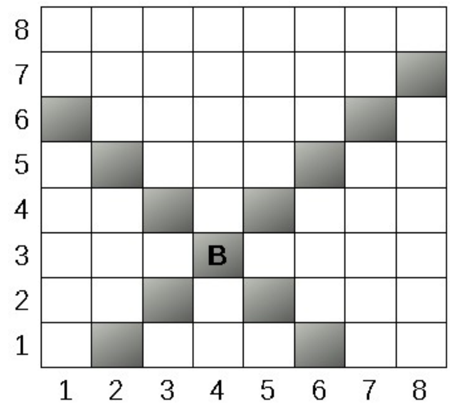
Here are some more examples:

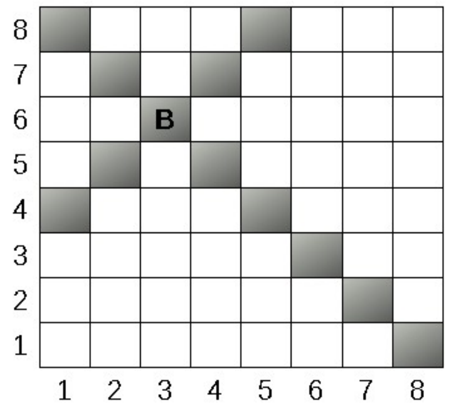
```
Numerator and denominator: 4 10
4 black and 2 white

Numerator and denominator: 2 13
16 black and 24 white

Numerator and denominator: 17 80
222 black and 259 white
```

6. Bishop Covers

In chess, a bishop can move any number of squares in any diagonal direction. The squares of a chessboard are colored black and white in an alternating pattern, but for this problem the colors do not matter so we will imagine bishops sitting on an all-white 8 x 8 grid.

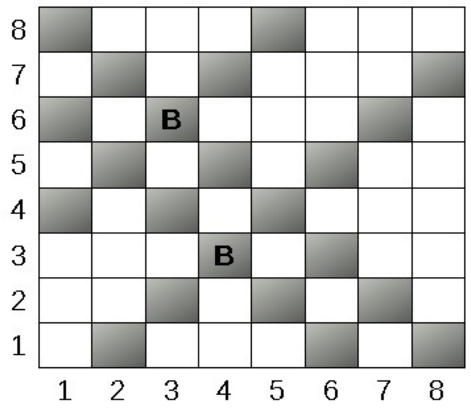
Let us say that a bishop *covers* the square on which it sits and all of the squares to which it could move. Here are two illustrations of the idea:





The bishop on the left is at position (4, 3) and the one on the right is at position (3, 6). The squares covered by each bishop are shaded.

Write a program that prompts the user to enter the positions of two bishops and outputs the total number of covered squares. For example, consider the bishops at positions (4, 3) and (3, 6) as shown above. Together they cover 22 squares:



The required I/O format is shown in the following execution snapshots:

```
Row and column of first bishop: 4 3
Row and column of second bishop: 3 6
22 squares covered.

Row and column of first bishop: 2 8
Row and column of second bishop: 7 5
17 squares covered.
```

7. Dromedary Words

A *dromedary* is a kind of camel with one hump. So let's call a sequence of letters a dromedary word if the letters strictly increase (in alphabetic order) up to some point in the word and after that point they strictly decrease.

For example, AFRICA is a dromedary word: A < F < R and I > C > A.

A single letter is considered to be both a strictly increasing and a strictly decreasing sequence, which means that SLED is a dromedary word since we can write S and L > E > D. By the same token, BEST is a dromedary word since we can write B < E < S and T.

On the other hand, CREAM is *not* a dromedary word: C < R and E > A, but the decreasing sequence cannot be extended with M.

Note that APPLE is *not* a dromedary word: we could try to break it down as A < P ≤ P and L > E, but the first part is not *strictly* increasing because of the repeated P.

Write a program that prompts the user for a word (all upper-case letters) and then tells the user if it is a dromedary word or not. The required I/O format is illustrated by the following execution snapshots.

```
Enter a word: AFRICA
AFRICA is a dromedary word.

Enter a word: GROKE
GROKE is a dromedary word.

Enter a word: AFILOTUXUNEA
AFILOTUXUNEA is a dromedary word.

Enter a word: QUAIL
QUAIL is NOT a dromedary word.
```

8. Parity Signature

The *parity* of an integer indicates whether it is even or odd. For example, 4327 has odd parity and 4328 has even parity.

Given an integer, we can produce its *digit parity sequence* by looking at the parity of each digit. We will use 0 to denote even and 1 to denote odd. For example, the digit parity sequence of 493,744,392 would be 011100110.

We can group digit parity sequences according to their *signature*. This idea is best explained by an example. The signature of 011100110 is 1, 3, 2, 2, 1 because the sequence starts with a single zero, then is followed by 3 ones, after which there are 2 zeros, and so on. You can picture it like this:

$$\begin{array}{ccccc} 0 & 111 & 00 & 11 & 0 \\ \hline 1 & 3 & 2 & 2 & 1 \end{array}$$

Write a program that prompts the user for two integers and outputs the parity signature of their product. You may assume that the product will not exceed two billion.

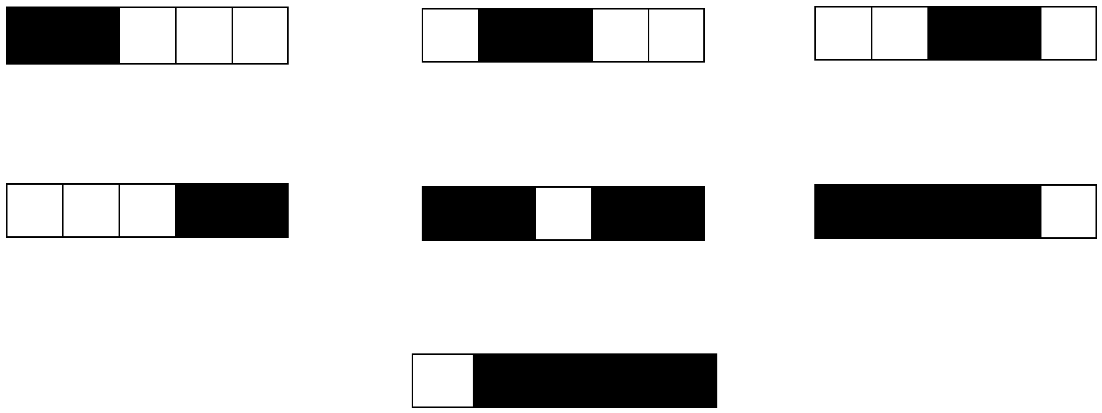
The required I/O format is illustrated by the following execution snapshots:

```
Enter two positive integers: 100 25
Parity signature of product 2500 = 1 1 2

Enter two positive integers: 11111 54321
Parity signature of product 603560631 = 2 2 3 2

Enter two positive integers: 2468 13579
Parity signature of product 33512972 = 0 4 1 2 1
```

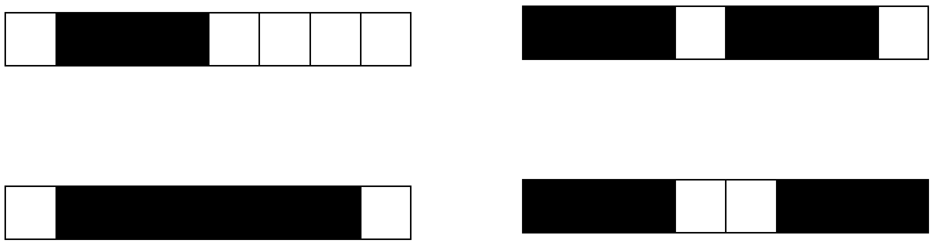
Note that the signature always begins with the count of even numbers at the beginning of the sequence. That's why the output in the last execution snapshot is 04121 instead of 4121.

9. Replacing Tiles

Imagine a row of five square white tiles. That is pretty boring, so imagine replacing some of the square tiles with 2 x 1 rectangular black tiles. There are 7 ways in which this might be done:



Now imagine a row of eight square white tiles and consider ways of replacing some of them with 3 x 1 rectangular black tiles. There are 12 ways to do this. Here are four possibilities:



Write a program that prompts the user for an integer representing the number of square white tiles and the length of the rectangular black tiles, and outputs the number of ways in which white tiles can be replaced with black ones. You may assume an upper limit of 25 square white tiles.

Execution snapshots:

```
Number of square white tiles: 8
Length of black tiles: 3
There are 12 replacement configurations.

Number of square tiles: 12
Length of replacement tiles: 4
There are 25 replacement configurations.

Number of square tiles: 10
Length of replacement tiles: 2
There are 88 replacement configurations.
```

10. Enemies

There are two kinds of people in this world, *glarks* and *flerbs*. When they go to the movies, a glark will sit next to anyone, but two flerbs will never sit next to each other.

In how many ways can a row of 5 seats be filled with glarks and flerbs? The answer is 13. Here are all the possible arrangements:

1	glark	glark	glark	glark	glark
2	flerb	glark	glark	glark	glark
3	glark	flerb	glark	glark	glark
4	glark	glark	flerb	glark	glark
5	glark	glark	glark	flerb	glark
6	glark	glark	glark	glark	flerb
7	flerb	glark	flerb	glark	glark
8	flerb	glark	glark	flerb	glark
9	flerb	glark	glark	glark	flerb
10	glark	flerb	glark	flerb	glark
11	glark	flerb	glark	glark	flerb
12	glark	glark	flerb	glark	flerb
13	flerb	glark	flerb	glark	flerb

Write a program that prompts the user for the number of seats in a row (1-30) and then outputs the number of ways in which glarks and flerbs can fill the seats in that row.

Execution snapshots:

```
How many seats: 3
Number of arrangements: 5

How many seats: 10
Number of arrangements: 144

How many seats: 25
Number of arrangements: 196418
```