A+ Computer Science
December 2014

Computer Science Competition
Hands-On Programming Set

I. General Notes

1. Do the problems in any order you like. They do not have to be done in order from 1 to 12.
2. All problems have a value of 60 points.
3. There is no extraneous input. All input is exactly as specified in the problem. Unless specified by the problem, integer inputs will not have leading zeros. Unless otherwise specified, your program should read to the end of file.
4. Your program should not print extraneous output. Follow the form exactly as given in the problem.
5. A penalty of 5 points will be assessed each time that an incorrect solution is submitted. This penalty will only be assessed if a solution is ultimately judged as correct.

II. Names of Problems

Number	Name
Problem 1	Banners
Problem 2	Chevron
Problem 3	Condensation
Problem 4	Continued
Problem 5	Equality
Problem 6	Hello Word
Problem 7	Moat Point
Problem 8	Stones
Problem 9	Survivor
Problem 10	Triangles
Problem 11	Wall Street
Problem 12	Word Pro

1. Banners

Program Name: banners.java Input File: banners.dat

Banners of all shapes and sizes hang along the wall in a school, celebrating the accomplishments of the students over the years, but occasionally changes have to be made. As old banners fade and wear out, they must be removed, and as new banners come around they need to be hung. Your job is to manage this process as efficiently as possible.

A "remove banner" order consists of the word "REMOVE" followed by an integer indicating which banner is to be removed from the wall. For example, "REMOVE 4" means to remove the 4th banner from the left, freeing up that space for other new banners to be hung later on. "ADD 12" means to find the first available space from the left to hang a new banner of size 12. You are to report the left side position of the banner that was removed, or of the banner that was hung. The position on the very far left of the wall is position 1. For example, initially you might have banners hung along the wall at the following positions and sizes - 5 2 (banner of size 2 hung at position 5), 16 11, 35 3, 38 4, 42 8, 51 6 - which results in an array of banners as shown below. A banner is shown with the \ character as the left edge, and * characters for the remaining units. A size 5 banner would be `\****`.

```
----\*---------\**********--------\**\***\*******-\*****----
                                     ^^^^
```

A "REMOVE 4" order would result in the 4th banner on the wall removed from position 38, leaving that space empty.

```
REMOVE BANNER AT POSITION 38
----\*---------\**********--------\**----\*******-\*****----
                                     ^^^^
```

Subsequent orders of "ADD 8" and "REMOVE 4" would result in the following:

```
ADD BANNER AT POSITION 7
----\*\*******-\**********--------\**----\*******-\*****----
      ^^^^^^^^
REMOVE BANNER AT POSITION 35
----\*\*******-\**********--------------\*******-\*****----
                                  ^^^
```

There is a limit of 100 units of wall space for all of the banners, but for the purposes of this program, there will always be room to add a new banner. Likewise, a remove order will always indicate the removal of an existing banner.

Input

A single line of ordered integer pairs indicating the starting position and size of each banner. This is followed by an integer N indicating N banner orders, each consisting of the word "REMOVE" or the word "ADD", and then followed by an integer. The integer after the "REMOVE" order indicates which banner currently hanging is to be removed. An integer after the word "ADD" indicates the size of the new banner to be hung in the first available open space.

Output

For each order, output a report of the action taken as shown in the examples above.

Sample Input File

```
5 2 16 11 35 3 38 4 42 8 51 6
3
REMOVE 4
ADD 8
REMOVE 4
```

Resulting Output

```
REMOVE BANNER AT POSITION 38
ADD BANNER AT POSITION 7
REMOVE BANNER AT POSITION 35
```

2. Chevron

Program Name: chevron.java **Input File: chevron.dat**

Chevrons have ancient meanings in history, and are still in use today, most notably for military rank insignias. The chevron occurs in early art including designs on pottery and rock carvings. A chevron is also one of the symbols used in heraldry, one of the simple geometrical figures which are the chief images in many coat of arms. When shown as a smaller size than standard, it is called a chevronel.

Your job is to create a ***chevron*** or ***chevronel*** that symbolizes the rank of corporal (one chevron), lance corporal (two chevrons stacked) or sergeant (three chevrons stacked). A chevron will be two character layers thick and seven characters tall, and a chevronel only one layer five characters tall.

For example, if the designation is ***sergeant chevron***, the symbol created should look like this:

```
**         **
 **       **
  **     **
** **   ** **
 ** ** ** **
  ** *** **
** ** * ** **
 ** ** ** **
  ** *** **
   ** * **
    ** **
     ***
      *
```

A ***corporal chevronel*** would look like this:

```
*       *
 *     *
  *   *
   * *
    *
```

Input

An initial value N with N sets of data to follow. Each data set consists of two words, **sergeant, corporal,** or **lance,** followed by the word **chevron** or **chevronel.**

Output

For each data set, output the corresponding chevron or chevronel, using the asterisk symbol (*), as shown in the examples. There is to be no blank separation lines between symbols.

Sample Input File

```
3
sergeant chevron
corporal chevronel
lance chevronel
```

Resulting Output

```
**         **
 **       **
  **     **
** **   ** **
 ** ** ** **
  ** *** **
** ** * ** **
 ** ** ** **
  ** *** **
   ** * **
    ** **
     ***
      *
*       *
 *     *
  *   *
   * *
    *
*       *
 *     *
* *   * *
 * * * *
  * * *
   * *
    *
```

3. Condensation

Program Name: condensation **Input File: condensation.dat**

There are many ways to compress a document into a smaller version, condensing it into a smaller size for transport or storage, and then decompressing it back to its original form afterwards. Below is a very simple compression algorithm, with only a few rules to follow.

Input

Given a word, compress it by applying the following rules (in order):

1. If the word is a number zero through nine, convert it to the corresponding digit.
2. If the word is shorter than four letters, leave it alone.
3. Change all consecutive runs of a letter to a single copy of the letter.
4. Remove all vowels (a,e,i,o,u), except if they start or end a word.

Input

Several words from data file, each one its own line. Each word will contain only letters - no symbols, digits, or spaces.

Output

The shortened version of each word, based on the rules listed above.

Sample Input File

```
bookkeeper
beehives
three
eat
elephants
```

Resulting Output

```
bkpr
bhvs
3
eat
elphnts
```

4. Continued

Program Name: continued.java **Input File: continued.dat**

A “continued fraction” is one that is written in the form: a positive integer plus a fraction whose numerator is 1 and whose denominator is a positive integer plus a fraction with a numerator of 1, and so on, and so on…

Here is an example of a continued fraction:

$$\frac{27}{8} = 3 + \frac{3}{8} = 3 + \frac{1}{8/3} = 3 + \frac{1}{2 + \frac{2}{3}} = 3 + \frac{1}{2 + \frac{1}{3/2}} = \mathbf{3} + \frac{1}{\mathbf{2} + \frac{1}{\mathbf{1} + \frac{1}{\mathbf{2}}}}$$

The continuation process stops when the numerators are all 1. The final significant values of this example are **3, 2, 1 and 2.**

Input

Several positive fractions represented as ordered integers, each pair on one line.

Output

The final significant integer values as shown in the example above, each separated by a single space.

Sample Input File

```
27 8
1 2
3 2
7 4
```

Resulting Output

```
3 2 1 2
0 2
1 2
1 1 3
```

5. Equality

Program Name: equality.java **Input File: equality.dat**

This is one is simple, if you know what is needed. Study it carefully and you will find a way.

Input

Several pairs of integers, each on a separate line, which indicate a pattern of numbers, which should be evident from the examples shown.

Output

For each pair of integers, output five equality statements as shown below, some of which are true, and some of which are false. Examine carefully the output pattern for each and determine the best way to solve this problem. It is easier than you think.

Sample Input File

```
90 10
-150 10
100 7
-130 4
```

Resulting Output

```
 +90 ==  +90 is true
+100 == +100 is true
+110 == +110 is true
+120 == +120 is true
+130 == +130 is false

-150 == -150 is false
-140 == -140 is false
-130 == -130 is false
-120 == -120 is true
-110 == -110 is true

+100 == +100 is true
+107 == +107 is true
+114 == +114 is true
+121 == +121 is true
+128 == +128 is false

-130 == -130 is false
-126 == -126 is true
-122 == -122 is true
-118 == -118 is true
-114 == -114 is true
```

6. Hello Word

Program Name: Hello.java Input File: none

Say "HELLO" to this packet, specifically using the first word of the problem description for each problem.

Input

None.

Output

Output twelve statements, each that say "`HELLO <FIRST WORD>`", using the first word from each of the twelve problem statements. For example, to say Hello to the first problem, you would output **`HELLO BANNERS`**. It may not make much sense, but it should be easy enough to do. Oh, and one more thing, the twelve sentences need to be in alphabetical order.

Example Output

```
HELLO BANNERS
HELLO CHEVRON
.
.
```

.

7. Moat Point

Program Name: moat.java Input File: moat.dat

In various card games, hands of cards are assigned point values based on various systems. One system is called Moat Points. The Moat Point value of a hand gives each Ace 15 points, each face card (Jack, Queen, King) 12 points, and all other cards their face value (for example, a 7 is worth 7 points).

For example, the following hand :

Clubs: 10,9
Diamonds: A,Q,6,5
Hearts: A
Spades: K, 10, 7

would be worth 101 Moat points (10+9+15+12+6+5+15+12+10+7).

Your job is to find the moat point value for each given hand.

Input

An initial value N with N hands to follow, each on one line. Each hand consists of a 20 character string representing 10 cards, a character for the value of the card (A, 2-9, 0 for the value 10, J, Q, K) and a letter for the suit (S for spades, D for diamonds, C for clubs, and H for hearts). Each hand is guaranteed to be dealt from a newly shuffled deck of cards.

Output

For each hand, print the moat point value, labeled as shown.

Sample Input File

```
4
0C9CADQD6D5DAHKS0S7S
4SQD6D4D0C9CJS6C6H2D
JD5H5C3S3H3C8SAS2H0H
2SQC0C0S7SAH5SACKHQH
```

Resulting Output

```
101
71
66
100
```

8. Stones

Program Name: stones.java Input File: stones.dat

An ancient game of solitaire is called "stones". A player selects a random number of stones and divides them up into a random number of piles. Each pile need not have the same number of stones to begin with. For example, a player may choose 12 stones and put them into 4 piles, containing 1, 6, 3, and 2 stones respectively. The player then makes a move, which consists of taking 1 stone from each pile, and making a new pile with those stones.

In this game, after the first move, there would be piles of 5, 2, 1, and 4 stones, each starting pile reduced by 1, and a new pile created by the 4 removed stones. The next move would result in piles of 4, 1, 3, and 4, then 3, 2, 3, and 4, then 2, 1, 2, 3 and 4, then 1, 1, 2, 3, 5, and finally 1, 2, 4, 5, which results in a loss in 6 moves.

The reason for the loss is that this last pile matches one of the previous resulting piles (not the starting one, and not the one right before the last move), namely the pile after the first move (5, 2, 1, 4 - order does not matter).

A win can only result if the number of stones after the next move would match exactly the number of stones in the current set of piles, for example, if 3, 2, and 1 are the piles after a move, the next move would result in 2, 1, and 3, an exact match, therefore a win at 3, 2, 1.

Here is the sequence in the games described above, with pile number sorted in descending order:
START 6, 3, 2, 1
Move # 1 - 5, 4, 2, 1
Move # 2 - 4, 4, 3, 1
Move # 3 - 4, 3, 3, 2
Move # 4 - 4, 3, 2, 2, 1
Move # 5 - 5, 3, 2, 1, 1
Move # 6 - 5, 4, 2, 1
A loss in 6 moves, with 5, 4, 2, 1 matching the hand after Move #1

START [3, 2, 1]
Move # 1 [3, 2, 1]
Move # 2 [3, 2, 1]
A win in 1 move, with 3, 2, 1, Move #2 would match the result after Move #1

Input

An initial value N with N sets of data to follow, each on one line. Each set of data consists of several integers, the first two of which represent the total number of stones in the game, the number of piles made, and the number of stones in each pile.

Output

For each game, output the result of the game as shown below, with the number of moves, the word WIN or LOSS, followed by the contents of the hand at the end of the game, in descending sorted order, exactly as shown with single spaces of separation.

Sample Input File

```
2
12 4 1 6 3 2
28 5 2 8 4 5 9
```

Resulting Output

```
6 LOSS [5, 4, 2, 1]
17 WIN [7, 6, 5, 4, 3, 2, 1]
```

9. Survivor

Program Name: survivor.java **Input File: survivor.dat**

Survival of the fittest over time and death is a natural rule of evolution, and this problem is a simulation of that process, based on a very simple life model. In an 8X8 grid, similar to one used in chess or checkers, a cell indicating an organism contains a '*', with empty cells designated by a period '.' Each cell interior to the board has 8 bordering cells that are either empty or that contain an organism. Cells on the edge of the board have 5 bordering cells, and corner cells have 3 bordering cells. Each bordering cell that contains an organism is called a **neighbor**.

Survival of the entire family of organisms on the board depends on how many neighbors each cell has. To survive to the next generation, each organism must have only 2 or 3 neighbors. Any more would cause that single organism to die due to over-crowding, and any fewer would cause death from isolation. Furthermore, a new organism is "born" into the next generation in an empty cell that has EXACTLY 3 current neighbors.

To summarize:

- An organism cell with too few (less than 2) or too many (4 or more) neighbors becomes empty in the next generation, otherwise it survives.
- An empty cell that has EXACTLY 3 neighbors is "born" and becomes an organism in the next generation.

Below is an example. Four organisms are shown for generation zero. For the next generation, the first and fourth ones die due to isolation. The second and third survive since each has two current neighbors. The empty cells above and below the middle two spawn new organisms since each one has exactly three current neighbors.

The process continues to Generation Two as shown in the third grid.

The top left cell of the board will be considered row 1, column 1. The organisms shown in generation zero at are positions (4,4), (4,5), (4,6), and (4,7).

```
Generation Zero                Generation 1                   Generation 2
........                       ........                       ........
........                       ........                       ........
........                       ....**..                       ....**..
...****.                       ....**..                       ...*..*.
........                       ....**..                       ....**..
........                       ........                       ........
........                       ........                       ........
........                       ........                       ........
```

Input

An initial value N with N sets of data to follow, each on one line. Each set of data consists of an integer P, with P ordered pairs of integers (R, C) to follow, each designating a zero generation organism located at position (R,C) on the board.

Output

For each zero generation given, output generations zero, one, and five, as shown in the example above, with one blank line of separation after each grid.

Sample Input File

```
2
4 4 4 4 5 4 6 4 7
8 1 1 2 2 3 3 4 4 5 4 6 3 7 2 8 1
```

Resulting Output

```
GENERATION ZERO
........
........
........
...****.
........
........
........
........

GENERATION 1
........
........
....**..
....**..
....**..
........
........
........

GENERATION 5
........
........
....**..
...*..*.
....**..
........
........
........

GENERATION ZERO
*.......
.*......
..*.....
...*....
...*....
..*.....
.*......
*.......

GENERATION 1
........
.*......
..*.....
..**....
..**....
..*.....
.*......
........

GENERATION 5
........
.**.....
..*.....
........
........
..*.....
.**.....
........
```

10. Triangles

Program Name: triangles.java **Input File: triangles.dat**

Given the three ordered pair coordinates of a triangle, output its area and perimeter rounded to the nearest whole number. A useful formula for area of any triangle is Heron's formula, which is as follows:

$$\sqrt{s(s-a)(s-b)(s-c)}$$

where s is the semi-perimeter of the triangle, and a, b, and c are the three side lengths.

Input

Several lines of data, each consisting of three ordered pairs of integer values corresponding to the three points of a triangle on a coordinate plane.

Output

The area and perimeter of the triangle, rounded to the nearest whole number.

Sample Input File

```
20 20 35 20 20 30
-12 3 9 31 20 -17
0 0 3 0 3 4
```

Resulting Output

```
75 43
658 122
6 12
```

11. Wall Street

Program Name: wall.java Input File: wall.dat

Wall Street is well-known as home of the New York Stock Exchange, arguably the king of all world stock markets. Millions of shares of stock are traded with buy and sell orders issued by stock owners across the globe.

In this problem we will explore a very simplistic model of buying and selling stocks. Given your initial purchase of shares in the stocks four companies A, B, C and D, followed by your transaction orders at whatever increase or decline in the share price, your job is to report the total value of the portfolio at the original time of purchase, the number of shares left in each company after your transaction orders, as well as the total value of your portfolio immediately after the transaction orders are executed.

For example, your initial order might be 75 shares of Company A selling for $20 per share, 100 shares of Company B at $42, 50 of C at $6, and 25 of D at $154. After some price fluctuation you make transaction orders to buy 10 more shares of Company A, which has increased in price by $1, 30 more of B at $3 less, dump 40 shares of C at $5 more per share, and acquire 10 more D shares at $100 less. The initial value of the portfolio is 75x$20 + 100x$42 + 50x$6 + 25x$154 = $9850. After the transaction orders, the portfolio value is $8855.

Assume you have deep pockets with no limits of what you can purchase.

Input

An initial value N with N sets of data to follow. Each set of data consists of two lines of four ordered pairs of integers, each pair representing the number of shares involved in the transaction followed by the price per share for that transaction. The pairs on the first line show your original purchase in the four companies, and the ones on the second line represent your transaction order, with the change in price of each share. A decline in price will never go below $1 per share, but you might accidentally order a sale of more shares than you own, at which point all of your shares of that company are sold, resulting in zero shares for that company.

Output

For each data set, output all on one line separated by single spaces the following items:

- total value of the portfolio after the original purchase
- the number of shares in each of the four companies after the transaction orders, and
- the value of the portfolio after the transaction orders are executed.

Sample Input File

```
2
75 20 100 42 50 6 25 154
10 1 30 -3 -40 5 10 -100
50 1 85 2 45 3 15 4
15 1 -100 2 23 -2 -14 10
```

Resulting Output

```
9850 85 130 10 35 8855
415 65 0 68 1 212
```

12. Word Pro

Program Name: word.java Input File: word.dat

Before word processing became the primary tool of writers, but after the advent of the typewriter, some writers would still write by hand, then hire a **"word professional"**, or typist, to type out their work. A typical scenario for such a job would be for the typist to charge a certain amount for the number of hand-written pages were submitted by the writer, and then another charge for the resulting number of typed pages.

For this problem, we will assume that on average every 3 hand-written pages will become 2 typed pages. The typist charges 10 cents for each hand-written page, and 35 cents for each resulting typed page. More often than not the last typed page will not be a full page, but it still counts as a full page in the cost calculation.

For large jobs, the typist offers three discounts. First, if there are more than 50 resulting typed pages, the first 50 cost 35 cents each and any beyond that are only charged 27 cents each. Second, if there are more than 40 hand-written pages to be typed, any pages beyond the first 40 are only charged 9 cents each, with the first 40 still at the normal 10 cent rate. Finally, if there are 100 or more hand-written pages, all of those pages are only charged at 8 cents per page.

For example, 85 hand-written pages would cost $27.44, calculated like this: 40 pages at 10 cents each is $4.00 + 45 pages at 9 cents is $4.05 + 57 typed pages at $17.50 for the first 50 and $1.89 for the remaining 7 pages.

Input
The date file will contain several integers. Each integer represents some number of hand-written pages.

Output
Print the cost of each typing job, in dollar format. Format the output to 2 decimal places.

Sample Input File
```
85
34
256
```

Resulting Output
```
$27.44
$11.45
$70.65
```