Computer Science Competition Invitational A 2020 Programming Problem 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	Ambrocio
Problem 2	Brian
Problem 3	Emilia
Problem 4	Igor
Problem 5	Jason
Problem 6	Jose
Problem 7	Keith
Problem 8	Lukas
Problem 9	Mauricio
Problem 10	Ram
Problem 11	Sebastian
Problem 12	Tatiana

1. Ambrocio

Program Name: Ambrocio.java **Input File: None**

Ambrocio is learning to program in Java, but often conflates the different symbols used for boolean operators. He's made a cheat sheet for himself to help keep each of the symbols straight.

Write a program that replicates Ambrocio's cheat sheet.

Input: There is no input for this problem.

Exact Output:
```
Programming uses lots of symbols I don't normally use.
& means AND
| means OR
^ means XOR
And don't forget to end with a semicolon (;)
```

2. Brian

Program Name: Brian.java Input File: brian.dat

Sun Microsystems released the first version of Java in 1995. Since then, Java has become one of the most widely-used programming languages worldwide, seeing frequent use in both academia and industry. The language has evolved continuously since then. Java 5 introduced generics, which made it possible to write more general data structures, and Java 8 introduced lambda functions. As of September 2019, the latest version of Java is Java 12.

Brian is a Java programmer, and wonders what version of Java he'll be using in the future. To simplify his predictions, he assumes that Java 0 came out in 1995, Java 12 came out in 2019, and that Java has a regular release cycle. Given a year, can you calculate what version of Java he'll be using?

Input: The first line of input has an integer T, the number of test cases. Each test case has a single line with a four-digit number, the year Brian is interested in. All years will be greater than or equal to 1995, but before the year 3000.

Output: For each test case, output the line "**`In the year YEAR, Brian will be coding in Java X!`**", where YEAR is the value given as input and X is the version of Java he expects to use. Use future tense even if the year is in the past.

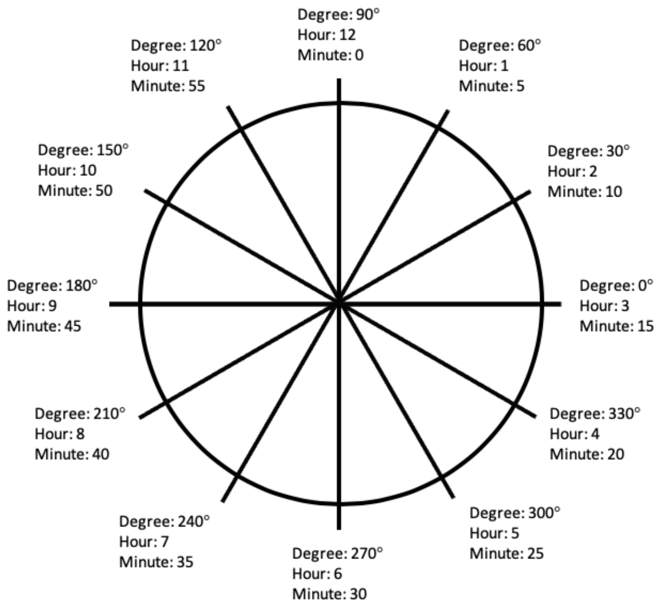
Sample Input:
```
5
1995
2019
2010
2025
2100
```

Sample Output:
```
In the year 1995, Brian will be coding in Java 0!
In the year 2019, Brian will be coding in Java 12!
In the year 2010, Brian will be coding in Java 7!
In the year 2025, Brian will be coding in Java 15!
In the year 2100, Brian will be coding in Java 52!
```

3. Emilia

Program Name: Emilia.java **Input File: emilia.dat**

Emilia just learned about the unit circle in her pre-calculus class. As she was leaving class she realized that the hour and minute hands of an analog clock not only are pointing to their respective hour and minute (representing the current time), but they are also pointing to a degree in range of [0,359] corresponding to the unit circle representation (See below figure). Emilia would like help writing a program that takes in two real numbers representing degrees, one for hour, and one for minute, and emits the corresponding time. Can you help her with this?



Input: Input starts with a line containing an integer N (1 <= N <= 10), the number of test cases. The following N lines, each with two degrees of floating point type, representing the hour degree and minute degree respectively. Hour degree and minute degree will be guaranteed to be in range of 0 <= value < 360.

Output: For each test case, output the corresponding time in the format of HH:MM

Sample input:

```
6
197.5 300.0
171.0 342.0
87.5 60.0
0.0 90.0
359.5 84.0
352.5 0.0
```

Sample output:

```
08:25
09:18
12:05
03:00
03:01
03:15
```

4. Igor

Program Name: Igor.java Input File: igor.dat

It's a long, hard day at the palindrome recycling factory. Workers take old unwanted strings, and recycle them into new palindromes! A palindrome is a string that reads the same forwards and backwards. For example, "racecar" and "noon" are palindromes, while "bus" and "midnight" are not.

To recycle a string S, first rearrange the letters of the string arbitrarily, then cut the string into pieces such that each piece is a palindrome. For example, given the string "atlanta", one possible way is to rearrange the letters into "lanatat" and then break them up to form the palindromes "l", "ana", and "tat". Note that these palindromes do not have to be valid English words.

The laziest solution to this is to break a string into individual characters, but the masses pay a premium for longer palindromes. The value of a palindrome P is equal to the square of its length. That is, "racecar" has a value of 49, and "noon" has an example score of 16.

Igor has been given a string S to recycle. What is the maximum value he can get from its parts?

Input:
Input starts with a line containing an integer T (1 <= T <= 20), the number of test cases. Following this are T lines, each with a single string S. The length of S is at least 1 and at most 2,000. Every character in S is a lowercase letter a-z.

Output: For each test case, output the case number and the maximum amount you can recycle this string for, formatted as in the samples.

Sample input:
```
4
atlanta
abcd
abcdefghijklmnopqrstuvwxyz
fwgqtsiczjnbavkmrpeyluhxdo
```

Sample output:
```
27
4
26
26
```

5. Jason

Program Name: Jason.java Input File: jason.dat

Jason has been working with parabolas in math class and has asked your programming team to make a program to help him check his homework. He will provide a number of sets of values for coefficients where each set defines a different parabola. For now, all he wants is to know the coordinates of the vertex and whether the parabola opens upward or downward. He provided the following explanation just in case you were not familiar with parabolas.

- Parabolas are the graphical depictions of quadratic functions of the form $f(x) = ax^2 + bx + c$ where the coefficients a, b, and c are real numbers that control the shape and placement of the parabola with $a \neq 0$.
- Parabolas are "bowl" shaped curves if graphed with good scales. The "bowl" opens either upward or downward with the bottom or top of the "bowl" located at the vertex which is the minimum or maximum point of the parabola.
- The vertex of a parabola is found at $x_v = \frac{-b}{2a}$ and its function value is just $f(x_v)$.
- When a > 0 the parabola opens upward and when a < 0 the parabola opens downward.
- First sample below: a = 2, b = 1, and c = 5.
  Vertex is at $\frac{-1}{2\times 2} = -0.250$ and $f(-0.250) = 2 \times (-0.250)^2 + 1 \times (-0.250) + 5 = 4.875$

Write a program for Jason to produce a list of vertex coordinates for a given set of parabola coefficients.

Input: First line of data file will contain a count N of the number of sets of coefficients that will follow. The next N lines will contain three whitespace separated decimal values for the coefficients *a*, *b*, and *c*. The maximum range of values for all coefficients will be ±99,999.99 and *a* will never be 0 but the others may be 0.

Output: A list of vertex coordinates in the form `(±99,999.999,±99,999.999)` followed by "`-->`" and the word "UPWARD" or "DOWNWARD" with each parabola on a separate line. Values > 0 will not display a + and there are no intervening spaces.

Sample input:
```
5
2 1 5
-5.3 7.2 -2.6
987.65 -123.45 -543.21
0.03 98.76 -91827.36
-3.1 0.0 0.0
```

Sample output:
```
(-0.250,4.875)-->UPWARD
(0.679,-0.155)-->DOWNWARD
(0.062,-547.068)-->UPWARD
(-1646.000,-173106.840)-->UPWARD
(0.000,0.000)-->DOWNWARD
```

6. Jose

Program Name: Jose.java Input File: jose.dat

Vehicle Identification Numbers or VIN numbers are the unique "name" for each motor vehicle manufactured in the world. A VIN is made up of 17 digits or characters (except for O, I and Q). Each digit or character or group of digits and characters in the VIN has a special meaning. These include the manufacturer, year made, country made and several other features about the vehicle the VIN has been assigned to. The 9th position in the VIN is a check digit in the United States and Canada. That 9th digit or character is where the story of Jose begins.

Jose has just gone to work for the National Highway Traffic Safety Administration which is in charge of VINs in the United States. As the junior programmer in the group, Jose has been tasked with the job of generating the check digit for each of several million potential VINs.

The check digit is generated using this formula. Multiply the value of each digit or character times the weight of the position that digit or character occupies. Find the sum of the products then find the remainder when the sum is divided by eleven. If the remainder is 0 – 9 then that becomes the check digit. If the remainder is 10 then the check digit is X. The check digit is placed in the 9th position in the VIN.

The value of each digit is the digit itself i.e. the value of 5 is 5. The value of each allowable character is shown in this table :

A: 1	B: 2	C: 3	D: 4	E: 5	F: 6	G: 7	H: 8	
J: 1	K: 2	L: 3	M: 4	N: 5		P: 7		R: 9
	S: 2	T: 3	U: 4	V: 5	W: 6	X: 7	Y: 8	Z: 9

The gaps are for the characters that are not allowed and the last line is shifted to the right to preserve the balance of the table.

The weight of each position is shown in this table.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Weight	8	7	6	5	4	3	2	10	0	9	8	7	6	5	4	3	2

For example the weight of position 3 is 6 and the weight of position 14 is 5. The weight of position 9 is 0 to ensure that the check digit itself is not part of the sum of the products.

The check digit for 1C3CDFDH8ED721434 (which is 8) is calculated like this.

character	1	C	3	C	D	F	D	H		E	D	7	2	1	4	3	4
value	1	3	3	3	4	6	4	8		5	4	7	2	1	4	3	4
weight	8	7	6	5	4	3	2	10	0	9	8	7	6	5	4	3	2
product	8	21	18	15	16	18	8	80		45	32	49	12	5	16	9	8

8 + 21 + 18 + 15 + 16 + 18 + 8 + 80 + 45 + 32 + 49 + 12 + 5 + 16 + 9 + 8 = 360

360 / 11 is 32 with 8 left over. So, the check digit is 8.

Jose has a file full of incomplete VINs. Each one is missing the check digit. Jose's job is to calculate a check digit for each VIN and place it in the 9th position of the incomplete VIN and print each completed VIN. Of course, Jose is going to write a program to do the job. What would that program look like?

Input: The input file is composed of three sections, one for the table of values assigned to each allowable letter, another for the table of weights assigned to each position and finally a list of incomplete VINs. The first table will contain a line for each allowable letter where the letter is followed by a space and the value. The second table will have a line for each position in the VIN where each line contains the position number followed by a space and then the weight of that position. Both tables will be followed by one blank line. The tables will be followed by a number N that represents the number of incomplete VINs to be processed. The list of incomplete VINs will begin on the next line after N and each will be on a separate line. The 9th position of each VIN will be an underscore.

Output: A list of completed VINs each on a separate line.

(continued next page)

Jose - continued

Sample input: *(Shown in two columns)*

```
A 1
B 2
C 3
D 4
E 5
F 6
G 7
H 8
J 1
K 2
L 3
M 4
N 5
P 7
R 9
S 2
T 3
U 4
V 5
W 6
X 7
Y 8
Z 9

1 8
2 7
3 6
4 5
5 4
6 3
7 2
8 10
9 0
10 9
11 8
12 7
13 6
14 5
15 4
16 3
17 2

10
1C3CDFDH_ED721434
1N4AL3AP_DC290218
1FMJU1HT_FEF20545
1GNDU03E_YD370482
2T3JF4DV_BW140581
3C63R3KJ_EG192714
5TDBT48A_1S268358
5NPDH4AE_DH252374
1LNHL9DR_AG613620
JAACR16E_J7234004
```

Sample output:

```
1C3CDFDH8ED721434
1N4AL3AP1DC290218
1FMJU1HT4FEF20545
1GNDU03E5YD370482
2T3JF4DV0BW140581
3C63R3KJ3EG192714
5TDBT48A61S268358
5NPDH4AEXDH252374
1LNHL9DR9AG613620
JAACR16E9J7234004
```

7. Keith

Program Name: Keith.java **Input File: keith.dat**

Keith has been talking about his physics class and how they are learning about velocity and acceleration. Anything that is moving, whether slowly or quickly, would have a velocity. That velocity might actually change over time like an automobile traveling at different speeds and in different directions. Technically, in physics, a velocity involves both speed and direction but let's ignore the direction for now and just focus on the speed as if it was constant and movement was in a straight line. The average speed is just the ratio between the distance and the period of time spent moving. Here are some examples:

- A 247 mile automobile trip between two cities that takes 3.5 hours would have an average speed of
  247 miles / 3.5 hours = 70.57 miles/hour
- Earth is on average about 92,960,000 miles from the sun and light from the sun takes about 8.3 minutes to reach the earth so the average speed of travel for the light is
  92,960,000 miles / 8.3 minutes = 11,200,000.00 miles/hour
- A jogger that completes a 10K (10000 meters) course in 73 minutes would have an average speed of
  10000 meters / 73 minutes = 136.99 meters / minute
- A Texas downpour that produces 3.7 inches of rain in 25 minutes is raining at the rate or speed of
  3.7 inches / 25 minutes = 0.15 inches / minute

Keith had a homework assignment in which each problem provided a distance and a time period, most with different units of measure. He knows you are part of a programming team and thought you could write a program to perform the computations so he could check his work. There are no unit conversions so it is just one number divided by another number and Keith just wants the raw speed without any units. However, you decided to add an extra twist and also identify the smallest and largest speeds of those provided.

Write a program to produce a list of speeds and identify the smallest and largest values.

Input: An unknown number of pairs of distance and time values with no units of measure. Each pair will be on a separate line and be separated by whitespace. Both will be non-zero and positive and may have decimal points. The maximum distance will be 99,999,999.99 and the maximum time will be 999,999.99.

Output: A list of speeds, one per line, with two decimal places of accuracy. Followed by two additional lines containing the smallest and largest values formatted as shown below.

Sample input:
```
247 3.5
92960000 8.3
10000 73
3.7 25
60910004.97 749.19741
```

Sample output:
```
70.57
11200000.00
136.99
0.15
81300.34
Min = 0.15
Max = 11200000.00
```

8. Lukas

Program Name: Lukas.java Input File: lukas.dat

Lukas just learned the rules for declaring a variable in Java. His textbook states that the following rules must be followed:

1. Variable names must start with a letter, the underscore (_) character, or $.,
2. The remaining characters must be letters, numbers, or underscores. (Technically, the $ symbol is allowed as well, but it is not recommended to use it—it is intended for names that are automatically generated by tools.) For the purposes of this program, $ will be accepted.
3. You cannot use other symbols such as ? or %, or any symbol located on the digit keys of a standard keyboard. Spaces are not permitted inside names either. You can use uppercase letters to denote word boundaries, as in cansPerPack. This naming convention is called *camel case* because the uppercase letters in the middle of the name look like the humps of a camel.)
4. Variable names are **case sensitive**, that is, canVolume and canvolume are different names.
5. You cannot use **reserved words** such as double or class as names; these words are reserved exclusively for their special Java meanings. These words are:
   abstract, assert, boolean, break, byte, case, catch, char, class, const, continue, default, double, do, else, enum, extends, final, finally, float, for, goto, if, implements, import, instanceof, int, interface, long, native, new, package, private, protected, public, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, try, void, volatile, while
6. The following are literals that you cannot use as variables name in you program:
   true, false, null

Lukas would like to write a program that checks whether a given variable name is in fact valid in Java, can you help him with this?

Input: Input begins with a **single long line** containing all reserved words from Rule 4. Words will be separated by a comma only (no spaces). The **second line of input is an integer N** (1 <= N <= 20), the number of test cases. The following N lines, each contain a variable name that your program will check to determine if the name is valid or not.

Output: For each test case, output Valid or Invalid. Valid if the name is valid name according to the criteria above, Invalid if the name is invalid according to the criteria above.

Sample input:

```
abstract,assert,boolean,break,byte,case,catch,char,class,const,continue,default,double,do,else,enum,extends,fin
al,finally,float,for,goto,if,implements,import,instanceof,int,interface,long,native,new,package,private,protect
ed,public,return,short,static,strictfp,super,switch,synchronized,this,throw,throws,transient,try,void,volatile,
while
10
yes
true
truE
$100
_thisIsAVariable
this_is_a_variable
number#
1switch
switch
dollar$
```

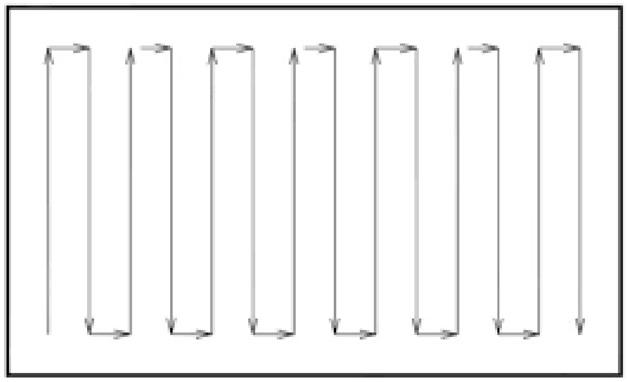
Sample output:

```
Valid
Invalid
Valid
Valid
Valid
Valid
Invalid
Invalid
Invalid
Valid
```

9. Mauricio

Program Name: Mauricio.java Input File: mauricio.dat

Mauricio's school just purchased a robotic lawnmower to mow all the rectangular patches of grass on their campus. Examples include the football field, soccer field, and the playground. The robotic lawnmower does not have an efficient path planning algorithm however, and will just roam around randomly until the rectangular region is fully covered. Mauricio's principal has asked him to implement a boustrophedon path plan to make the lawnmower more efficient in both time and energy. The Greek term boustrophedon translates to "the way of the ox". Today, boustrophedon means from right to left and from left to right in alternate lines. As the term implies the path plan requires the robot to traverse the full length of the field, turn around 180 degrees, mow the next portion of uncut grass, traverse back the full length of the field, turn around 180 degrees, and continue. This process is completed until the full area is covered. The below figure gives a visual example of a boustrophedon path plan.



Mauricio noticed that he could write an even better boustrophedon path plan that traversed the longest edge of rectangle first, versus the shortest edge. In traversing the longest edge first, his robot would not have to turn 180 degrees as much, reducing both time and energy! Can you help Maurico write a path planning program that implements a boustrophedon path plan that traverses the longest edge first?

Input: Input starts with a line containing an integer N (1 <= N <= 10), the number of test cases. The following N lines, each with two integer values representing the width W (number of rows) and the length L (number of columns) of the rectangular region to mow. The constraints for W and L are as follows:

$$2 \leq W \leq 70$$
$$2 \leq L \leq 70$$
$$L \neq W$$

Output: For each width W and length L, you are to print out the corresponding boustrophedon path plan that traverses the longest edge of the rectangular matrix first. Your path plan will incrementally number the order of which the elements of the rectangular regions are to be visited. Your output should be printed out in columns of D+1 size and right justified, where D is the number of digits of the last element visited. For example, if the last element visited was numbered 120, D would equal 3, so your output should be in columns of size 4 (D+1). Output numbering begins at 1, and always begins in the top, left element. Following each output set should be a line of 10 (ten) equal signs.

Sample input:

```
4
2 3
3 2
12 10
10 12
```

(continued next page)

Mauricio – continued

Sample output:

```
 1 2 3
 6 5 4
==========
 1 6
 2 5
 3 4
==========
   1  24  25  48  49  72  73  96  97 120
   2  23  26  47  50  71  74  95  98 119
   3  22  27  46  51  70  75  94  99 118
   4  21  28  45  52  69  76  93 100 117
   5  20  29  44  53  68  77  92 101 116
   6  19  30  43  54  67  78  91 102 115
   7  18  31  42  55  66  79  90 103 114
   8  17  32  41  56  65  80  89 104 113
   9  16  33  40  57  64  81  88 105 112
  10  15  34  39  58  63  82  87 106 111
  11  14  35  38  59  62  83  86 107 110
  12  13  36  37  60  61  84  85 108 109
==========
   1   2   3   4   5   6   7   8   9  10  11  12
  24  23  22  21  20  19  18  17  16  15  14  13
  25  26  27  28  29  30  31  32  33  34  35  36
  48  47  46  45  44  43  42  41  40  39  38  37
  49  50  51  52  53  54  55  56  57  58  59  60
  72  71  70  69  68  67  66  65  64  63  62  61
  73  74  75  76  77  78  79  80  81  82  83  84
  96  95  94  93  92  91  90  89  88  87  86  85
  97  98  99 100 101 102 103 104 105 106 107 108
 120 119 118 117 116 115 114 113 112 111 110 109
==========
```

10. Ram

Program Name: Ram.java Input File: ram.dat

The autocorrect on Ram's word processor has gone crazy! Every time Ram types something the word processor autocorrects it backwards. If Ram types **computer**, it is autocorrected to **retupmoc**. Ram has an essay due first thing in the morning and it is already midnight. Rather than deal with the autocorrect issue Ram decides to whip up a little program to read all of the backwards text he types and re-print it correctly. Write that program.

Input: A paragraph of text in which every word is printed backwards. A lines in the data file will have a length <= 80.

Output: The paragraph of text from the input file where every word is printed correctly. The output must match the number of lines in the input paragraph. There must be as many words on each line in the output as were in the input. If a punctuation mark lies within the word, it must be reversed along with the rest of the letters. If a punctuation mark precedes or follows the word, it must be left where it is. One punctuation mark will not follow another.

Sample input:
```
ZZ poT si na naciremA kcor dnab demrof ni 9691 ni notsuoH, saxeT.
ehT dnab sah, ecnis 0791, detsisnoc fo tsiratiug/tsilacov ylliB snobbiG,
eht s'dnab redael, niam tsiciryl dna lacisum regnarra, tsilacov/tsissab
ytsuD lliH, dna remmurd knarF draeB. "sA eniuneg stoor snaicisum, yeht
evah wef sreep" gnidrocca ot citirc leahciM "buC" adoK. snobbiG si eno
fo s'aciremA tsenif seulb stsiratiug gnikrow ni eht anera kcor moidi
elihw lliH dna draeB edivorp eht etamitlu mhtyhr noitces troppus.
```

Sample output:
```
ZZ Top is an American rock band formed in 1969 in Houston, Texas.
The band has, since 1970, consisted of vocalist/guitarist Billy Gibbons,
the band's leader, main lyricist and musical arranger, bassist/vocalist
Dusty Hill, and drummer Frank Beard. "As genuine roots musicians, they
have few peers" according to critic Michael "Cub" Koda. Gibbons is one
of America's finest blues guitarists working in the arena rock idiom
while Hill and Beard provide the ultimate rhythm section support.
```

11. Sebastian

Program Name: Sebastian.java **Input File: sebastian.dat**

Sebastian has decided that he is bored with the way words are alphabetized. He has come up with a scheme that places words in order based on the sum of the ASCII values of each of the characters in the words. For example the sum of the ASCII values in bravery is 763. The sum of the ASCII values in abracadabra is 1108. So, bravery would come before abracadabra in Sebastian's ordering scheme. Sometimes the sum of the ASCII values in two words are the same. When this is the case Sebastian just alphabetizes the two words. Sebastian has even come up with a name for his new alphabetizing method. He's calling it ASCIIbetizing. Clearly it is a lot more work to ASCIIbetize a list of words than it is to alphabetize them so Sebastian needs a program to do the job for him. That's where you come in. Write a program that will place a list of words into ASCIIbetized order.

Input: An unknown number of lines each containing an unknown number of words or numbers each separated by a space. There will be no punctuation within or between the words.

Output: An ASCIIbetized list of the words contained in the input file each on a separate line without duplicates. ***Despite the sample below shown in two columns, the actual output is to be all in one left aligned column.***

Sample input:

```
There was an Old Man with a beard
Who said It is just as I feared
Two Owls and a Hen
Four Larks and a Wren
Have all built their nests in my beard
```

Sample output: *(Shown in two columns)*

```
 I                                      was
 a                                      Have
 It                                     Four
 an                                     Wren
 as                                     said
 in                                     Owls
 is                                     with
 my                                     just
 Hen                                    There
 Man                                    Larks
 Old                                    beard
 Who                                    their
 and                                    built
 all                                    nests
 Two                                    feared
```

12. Tatiana

Program Name: Tatiana.java **Input File: tatiana.dat**

Tatiana's dad has been power walking almost every day for about a year to improve his fitness. She was telling him about the school's UIL programming team, and he had a brainstorm. Maybe she could get your team to write a program to crunch some of his fitness data. Modern fitness trackers like the iWatch can track a lot of health data but her dad decided to just focus on the daily distances he has been walking.

He wants a program that can process daily walking distances for anywhere from 1 month to 12 months. He will provide you with the distances walked for each day of the months to be processed. All he wants is a list of the total miles walked each month and how that month compares to an overall monthly average. To make the program flexible, the number of days in each month is also part of the data.

Impress Tatiana by writing a program to produce a walking distance report for her dad.

Input: First line of data file will contain a count N of the number of months of data that will follow. Each of the months will start with a line containing the number of days for that month which will be followed by that number of daily distances over several lines separated by whitespace. The number of months will not exceed 12 and the number of days in any month will not exceed 31. The distances will be non-negative and will not exceed 20.0 miles. Some days could be 0.

For the sample input, the number of days in the months will be small but judge's data will have months with normal numbers of days (1…31). The monthly sums are shown in sample output. The sum of those monthly sums is 84.13 for an average of 28.04 and the differences are: 1st month (01) 26.43 – 28.04 = -1.61, 2nd month (02) 31.57 – 28.04 = 3.53, and 3rd month (03) 26.13 – 28.04 = -1.91.

Output: One line for each month that starts with a month number (from 1…N) followed by the total distance walked that month and the distance that month is over (+) or under (−) the overall average of all months. The lines must be formatted precisely as shown below with leading zeros for month numbers < 10 and a leading plus or minus sign on the difference. There are exactly 3 spaces between fields and the two columns of distances must be capable of 3 digits to left of decimal and exactly 2 digits after the decimal.

Sample input:
```
3
5
5.36 5.03 5.31
5.06 5.67
6
5.04 4.15 4.17
6.95 6.23 5.03
5
2.13 6.32
6.96 6.22 4.50
```

Output line format (with spaces shown as carats ^)
```
99^^^999.99^^^±999.99
```

Sample output:
```
01    26.43     -1.61
02    31.57     +3.53
03    26.13     -1.91
```