Fifth Annual
University of Central Florida

High School Programming Tournament

Problems

Problem Name	Filename
The Alphabetizing Blues	BOOKS
Going in Circles	POLYGON
Parenthesis Numbering	PARENS
Raiders of the Last Arc	RAIDERS
Vandalism	VANDALS
Parity Checking	PARITY
Confused Choo Choo	TRAIN
Weird Multiplication	WEIRD
Sticks and Stones	NAMES

Call your program file: *Filename*.PAS or *Filename*.C
Call your input file: *Filename*.IN

For instance, if you are solving Sticks and Stones:

Call your program file: NAMES.PAS or NAMES.C
Call your input file: NAMES.IN

The Alphabetizing Blues

Filename: BOOKS

As the new computer programmer at the local library, your first assignment is to write a program to generate new labels for the bindings of worn-out books. The program must alphabetize the labels as they are generated, since the head librarian wants to see the labels in order as they will appear on the books once they are reshelved.

The Problem:

Given a list of books and their respective authors, alphabetize the books by title and print the list of titles and authors in a vertical format, as they might appear on their respective bindings. When alphabetizing, use **all** words in the books' titles--including "an" and "the"--even though that is not normally how books are alphabetized. No two books will have exactly the same title.

The Input:

Each book will be listed on a separate line with its author, in the format

title by *author*

Book titles will consist only of uppercase letters and spaces. The total number of characters on each line will be less than 80. There will be no more than 30 books listed.

The Output:

Print out the alphabetized list of books vertically, as though the books were next to each other on a library shelf. Each title/author should be separated from the next by one space. Follow the format shown in the Sample Output.

Sample Input:

```
WAR AND PEACE by Tolstoy
NEUROMANCER by Gibson
THE C PROGRAMMING LANGUAGE by Kernighan and Ritchie
CHAOS by Gleick
A BRIEF HISTORY OF TIME by Hawking
ALGORITHMS by Sedgewick
COMMON LISPCRAFT by Wilensky
THE TURING OMNIBUS by Dewdney
```

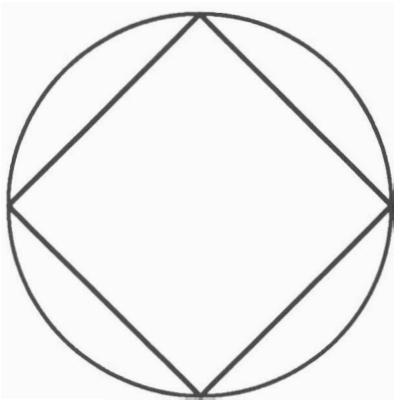
Sample Output:

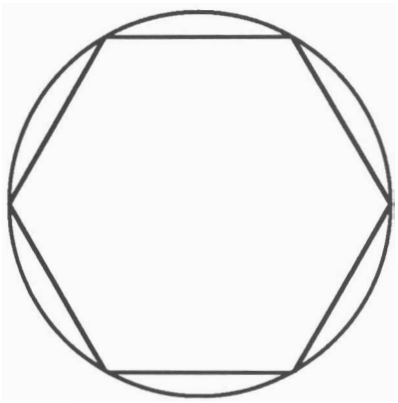
```
A ALGORITHM by Sedgewick
BRIEF HISTORY OF TIME by Hawking
CHAOS by Gleick
COMMON LISP CRAFT by Wilensky
NEUROMANCER by Gibson
THE PROGRAMMING LANGUAGE by Kernighan and Ritchie
THE TURING OMNIBUS by Dewdney
WAR AND PEACE by Tolstoy
```

Going in Circles

Filename: POLYGON

A polygon with all sides of equal length and all angles of equal measure is a **regular** polygon. If such a polygon is placed inside a circle so that the circle passes through each of its vertices, it is said to be **inscribed** in the circle. The following are regular polygons inscribed in circles:





The Problem:

Given an integer *n* and the radius of a circle r, compute the area of a regular polygon with *n* sides inscribed in that circle.

The Input:

There will be several lines in the input file. Each line will contain the integer *n* (where 2 < n < 20) followed by the radius of the circle *r*, a real number greater than zero.

The Output:

For each pair of values, print the message:

`Area = area`

where `area` is the appropriate computed value, rounded to 2 decimal places.

Sample Input:

```
4 14.1421
3 20
```

Sample Output:

```
Area = 400.00
Area = 519.62
```

Parenthesis Numbering

Filename: PARENS

The Problem:

Given a string containing only parentheses, your program is to determine whether or not the parentheses match and, if they match, indicate the corresponding ')' for each '('.

The Input:

Each string is on a separate input line. Each string will contain at least one and at most 26 parentheses. End of data will be indicated by end of file.

The Output:

For each input string in which the parentheses match correctly, print the string, placing two spaces in front of each parenthesis. On the next line, print a number for each left and right parenthesis to show the matching. The first '(' in an input string is numbered 1, the second '(' is numbered 2, the third '(' is numbered 3, and so on. Each ')' will have the same number as its matching '('. Print each number right justified in a 3 character field.

For each input string in which the parentheses do not match, print only the message

```
Parentheses do not match!
```

Leave a blank line after the output for each string.

Sample Input:

```
(()())(((())())())
(()
())
```

Sample Output:

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

Parentheses do not match!

Parentheses do not match!
```

Springfield Skippy *in....*

Raiders of the Last Arc

Filename: RAIDERS

Springfield Skippy teaches geometry at Yeehaw High. One of his students, Nat Malloy, became so disgusted with geometry that he plotted to hinder Skippy's ability to teach the subject. The evil Malloy managed to steal Skippy's most prized teaching aid: The Golden Circle. Nat divided it into 8 arcs and hid them throughout the school. Without the Golden Circle, Skippy is unable to continue teaching, and his students are doomed to failure. Skippy has managed to recapture 7 of the lost arcs, but he needs your help to find ... (ahem)... **The Last Arc!**

Malloy left behind a single clue which is rumored to indicate the position of the last arc. The clue is a staff with etchings that tell the directions to a secret hiding place. The translated instructions are as follows:

> *"Stand at the marked brick at the back wall of the gym. Take **m** paces directly away from the wall. Then take **n** paces north. Hold the staff vertically so it touches the floor. The sun shining through the front window will cause the staff to cast a shadow on the back wall of the gym. The tip of this shadow is the location of the last arc."*

The Problem:

Help Springfield Skippy find the last arc. You will be given *m*, *n*, and the length of the staff. Your program should determine the (x, y) coordinate location on the back wall where the last arc is hidden.

Assume that, when facing the back wall of the gym, you are facing west. Also, the gym wall extends infinitely upward, north, and south. The units for the coordinate axes on the back wall are feet. The marked brick is at location (0, 0) with respect to the wall, i.e. it is on the floor. The front window is directly across from the vertical axis of the back wall and is at a distance of 100 feet from the back wall and 50 feet up the front wall (the floor is horizontal, the front and back walls are parallel). You are guaranteed that the tip of the shadow will always fall on the back wall. The width of the staff is assumed to be negligible. Finally, Skippy knows that each pace he makes is 4 feet long.

The Input:

There will be several lines in the input file. Each line will contain values for *m*, *n*, and the length of the staff (in feet), respectively. The values on a given line will be separated from each other by single spaces. The values of *m* and *n* both will be integers; *m* will be greater than or equal to zero. A negative value given for *n* indicates a move south. The length of the staff will be a real number greater than zero.

The Output:

For each set of values in the input file, print the following message:

```
The last arc is at (x, y).
```

where (x, y) is the location of the tip of the shadow on the west wall. The units of x and y are feet, and the values should be rounded to two decimal places. Note that the coordinate axes are placed on the wall such that the origin is at the marked spot, the positive x-axis extends to the north, while the positive y-axis extends upward. Print each message on a new line.

Sample Input:

```
10 10 40.0
0 0 30.0
10 0 40.0
0 10 4.5
```

Sample Output:

```
The last arc is at (66.67, 33.33).
The last arc is at (0.00, 30.00).
The last arc is at (0.00, 33.33).
The last arc is at (40.00, 4.50).
```

Vandalism

Filename: VANDALS

There has been a rash of vandalism in our area. Vandals are stealing letters from outdoor letter signs and rearranging the remaining letters. The police have a list of suspects, all of whom (strangely enough) steal only the letters from their own names.

The Problem:

Given what a sign is supposed to say, what it now says, and a list of suspects, eliminate suspects based on the letters stolen. No thief will steal letters not in his own name, and no thief will steal more of a certain letter than appears in his name (e.g. Jim Black may steal one A, but never two A's). The thieves may, however, steal fewer letters than appear in their names, since they can get the other letters from other signs.

The Input:

The first line of input will contain an integer *p* indicating the number of lines in the original message. The next *p* lines will contain the original message. On the following line will be an integer *q* indicating the number of lines in the message after being vandalized. The next *q* lines will contain the message after being vandalized. On the following lines will be the names of suspects, one per line, until the end of the file. *p* and *q* will be between 1 and 10 (inclusive). The sign will always have at least one letter left on it. All messages and names will be composed exclusively of capital letters and spaces. There will be no lower case letters, numbers, punctuation or other characters. Spaces do not count as letters. Each line of the messages will be up to 70 characters long, and each name will be at least one but no more than 30 characters in length.

The Output:

For each suspect, print one of the following messages, whichever is appropriate:

```
name is still a suspect
name is no longer a suspect
```

where *name* is the suspect's name. Print each message on a new line.

Sample Input:

```
2
   WELCOME   TO   UCF
ALL HIGH SCHOOL PROGRAMMERS
1
WE    COME    TO   UCF   GRAMMAR   SCHOOL     PIGS ROLL
CHARLTON HESTON
CHARLES BOYER
FRED THE WONDER LLAMA
```

Sample Output:

```
CHARLTON HESTON is still a suspect
CHARLES BOYER is no longer a suspect
FRED THE WONDER LLAMA is no longer a suspect
```

Parity Checking

Filename: PARITY

Parity checking was developed to insure the accuracy of data bits (1's and 0's) transmitted from one computer to another. In one of the most common methods of parity checking, eight data bits (one byte) are transmitted, and then followed by a **parity bit**. The value of this parity bit depends on the values of the previous eight data bits as follows: If there are an *even* number of 1's in the transmitted byte, the parity bit will be 0. If there are an *odd* number of 1's in the transmitted byte, however, the parity bit will be 1, thus causing the total number of 1's for the nine-bit group (eight data bits, one parity bit) to be even. This method is called **even parity**.

The Problem:

Given several groups of eight data bits, determine the appropriate values for their respective parity bits, using even parity.

The Input:

There will be several lines in the input file. Each line will contain eight integers, separated by spaces. The integers will have values of either 0 or 1.

The Output:

For each line of data in the input file, print the values of the data bits represented, without spaces. Then print a space, followed by the appropriate parity bit (either a 1 or a 0). Print each nine-bit group on a new line.

Sample Input:

```
1 0 0 1 0 0 1 0
0 0 0 0 1 1 1 1
1 1 0 1 1 1 0 1
1 1 0 1 1 1 0 0
```

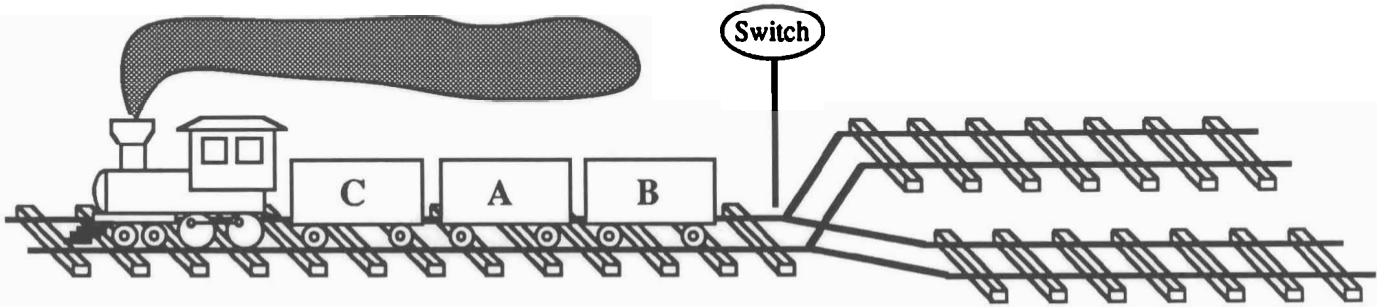
Sample Output:

```
10010010 1
00001111 0
11011101 0
11011100 1
```

Confused Choo Choo

Filename: TRAIN

A train is sitting on a track in the railyard with boxcars full of goodies to be delivered to different parts of the country. Unfortunately, the cars of the train are mixed up--the order in which they were connected to the engine does not match the order on the delivery invoices. To insure that the train will run on its optimum delivery schedule, the cars must be unscrambled by a process of unlinking cars on the different branches and relinking them in the correct order. There is a switch in the track which will connect the main track to either of the two parallel branch tracks, allowing the train to pass between them.



In the above diagram, C, A, and B each represent a car in the scrambled-up train. The correct order of cars is the alphabetic order from the *engine* to the *rear*.

The Problem:

By moving the train, switching between branch tracks, and unlinking and relinking cars, figure out how to put the train in the correct order (alphabetic from engine to rear). You will use the following commands:

reverse: Moves the train from the main track (on the left) to whichever of the branch tracks (on the right) to which the switch is currently set. The engine must be on the main track in order to execute this command.

forward: Moves the train from either of the branch tracks back to the main track. The engine must be on a branch track to execute this command.

switch: Changes the switch from one branch track to the other. The engine must be on the main track to execute this command.

couple: Reconnects *all* cars that have been unlinked on the current branch track to the rear of the mobile part of the train. The engine must be on a branch track to execute this command.

decouple at *X*: Disconnects the section of the train starting at *X* from the mobile part of the train, where *X* is a letter representing a car. The engine must be on a branch track to execute this command. For example, in the above train, **decouple at C** would separate cars **C**, **A** and **B** from the engine.

The train always starts on the main track, and when its cars are in the correct order, it should end up on the main track.

The Input:

There will be several lines in the input file. Each line will contain a string of unique uppercase letters. This string represents the cars of a train, listed in their current order from the engine to the rear. There will be at least one car but no more than six cars represented on each line.

The Output:

Print out the commands necessary to put the letters of each train in order, as shown in the Sample Output. List each command on a separate line, with no intervening blank lines. Print each command with all lower case letters, but capitalize the car letter in the **decouple at** command. Print no extra spaces before or in any of the commands.

When the train is in order and on the main track, print the message:

```
The train is ready to go!
```

Leave one blank line between solutions for different trains.

Note: There may be more than one correct solution for a particular train. Your program should only list one (not necessarily the shortest).

Sample Input:

```
CAB
MLK
```

Sample Output:

```
reverse
decouple at A
forward
switch
reverse
decouple at C
forward
switch
reverse
couple
forward
switch
reverse
couple
forward
The train is ready to go!

reverse
decouple at K
forward
switch
reverse
decouple at M
forward
switch
reverse
couple
forward
switch
reverse
couple
forward
switch
reverse
decouple at L
forward
switch
reverse
decouple at M
forward
switch
reverse
couple
forward
switch
reverse
couple
forward
The train is ready to go!
```

Weird Multiplication

Filename: WEIRD

Consider two positive (greater than zero) integers. Let *a* be the first number, and *b* be the second number. The following is a weird way to multiply these two numbers: repeatedly divide *a* by 2, and truncate the result, and double *b* at the same rate. Each time that *a* is odd, remember the value of *b*. Continue this process until *a* is 1. The sum of the remembered values of *b* will be the product of *a* and *b*. For example, to multiply 45 by 18:

Step 1: $a = 45$,	$b = 18$	*a* is odd, so remember 18
Step 2: $a = 45 / 2 = 22$,	$b = 18 * 2 = 36$	*a* is even
Step 3: $a = 22 / 2 = 11$,	$b = 36 * 2 = 72$	*a* is odd, so remember 72
Step 4: $a = 11 / 2 = 5$,	$b = 72 * 2 = 144$	*a* is odd, so remember 144
Step 5: $a = 5 / 2 = 2$,	$b = 144 * 2 = 288$	*a* is even
Step 6: $a = 2 / 2 = 1$,	**$b = 288 * 2 = 576$**	***a* is odd, so remember 576**

Therefore,

$45 * 18 = 18 + 72 + 144 + 576 = 810$

The Problem:

Given two integers greater than zero, perform this weird form of multiplication.

The Input:

There will be several lines in the input file. Each line will contain two integers to be multiplied, *a* (the first integer) and *b* (the second integer). Both numbers will be greater than zero.

The Output:

For each pair of numbers in the input file, print the following message:

$a * b = r_1 + r_2 + r_3 + \ldots + r_q = p$

where *a* and *b* are the input integers specified above, *q* is the number of remembered values, r_i (for values of *i* from 1 to *q* inclusive) are the remembered values, and *p* is the final product. You are guaranteed that the final product will be less than or equal to 32767.

Sample Input:

```
45 18
308 7
```

Sample Output:

```
45 * 18 = 18 + 72 + 144 + 576 = 810
308 * 7 = 28 + 112 + 224 + 1792 = 2156
```

Sticks and Stones

Filename: NAMES

So names will never hurt you, eh? Tell that to the poor souls who will have to remember and write their names years from now if a certain dangerous trend continues--to wit, husbands and wives combining their last names when they get married. Imagine: Mr. Smith and Ms. Doe get married, combine their names, and have children with the last name Doe-Smith. One of the Doe-Smith children grows up and marries someone with the last name of Wright-Jones. They combine their names and have children with the last name Doe-Smith-Wright-Jones. Where will it end?

The Problem:

Given a combined last name, determine the marriages of past generations which led to the formation of that name. The progenitors who have **simple names** prior to marriage (i.e. single-word, unhyphenated names) will all be in the first generation. Each generation thereafter will consist of people with **compound names** (i.e. two or more simple names, combined by hyphenating). Furthermore, people of a given generation will only marry others who have the same degree of "compoundedness" to their name. For example, a Doe-Smith will not marry a Bernstein or a Kennedy-Onassis-Trump-Quayle, but might marry a Chang-Martinez. Effectively, this will double the "compoundedness" of the names at every generation.

The Input:

The input will consist of a list of compound names, one per line. The names are guaranteed to consist of at most 80 characters, and will contain only upper and lower case letters and hyphens. No compound name will contain more than 16 simple names. Each simple name will contain at least one character.

The Output:

For each compound name given, print a record of the marriages, generation by generation, which led to the formation of that name. Consider all of the simple names to be from generation 1, and print the record in ascending order of generation number. For each marriage, print the following message:

`Generation` *`n`*`: A` *`name1`* `marries a` *`name2`*

After printing the marriages, print a message acknowledging the current generation as follows:

`Generation` *`n`*`: The` *`name`* `children`

where *n* is the generation number, and *name1*, *name2*, and *name* are the appropriate names. Print a blank line between output sets. Within a given generation, the order of marriage messages is not important.

Sample Input:

```
Throckmorton-Gingrich-Hamilton-Forbes
Alder-Zachary
```

Sample Output:

```
Generation 1: A Throckmorton marries a Gingrich
Generation 1: A Hamilton marries a Forbes
Generation 2: A Throckmorton-Gingrich marries a Hamilton-Forbes
Generation 3: The Throckmorton-Gingrich-Hamilton-Forbes children

Generation 1: A Alder marries a Zachary
Generation 2: The Alder-Zachary children
```