Eighth Annual
University of Central Florida

ACM • UPE

High School Programming Tournament

Problems

Problem Name	**Filename**
Back Seat Driver	DRIVER
Where's the Bee?	BEEHIVE
Vocal Punctuation	VOCAL
Dimwit Numbers	DIMWIT
Snail	SNAIL
In Line at the Bank	BANK
Base Alphabets	BASE
Nothin' But Net	NOTHIN
Visual Acuity	ACUITY

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

For example, if you are solving Where's the Bee?:

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

Back Seat Driver

Filename: `DRIVER`

Have you ever noticed that when driving with a group, the passengers in the back seat constantly shout conflicting directions? Someone will try to give directions from a map. At the same time, someone else will try to give directions from a street sign. Some people just shout directions for no good reason.

The Problem:

Given a string of garbled speech from the back seat, determine who is saying what and which directions to follow. The garbled sentence is a concatenation of pieces of the individual sentences spoken simultaneously by the people in the back seat. The sentence pieces are separated by the `!` character. That means that all characters before the first `!` are spoken by the first person, all characters between the first `!` and the second `!` are spoken by the second person, and so on. If there are more sentence fragments than back seat passengers, the fragments start back with the first person again.

The character combination `!!`, however, does not denote a change in speakers, but is used to denote that a single `!` is "spoken" by the current speaker; this is a way of measuring how loudly the current speaker is shouting. Of course, whoever yells the loudest (i.e., whoever "speaks" the most `!` characters in his sentence) is the person whose directions should be followed.

The Input:

There will be several data sets. Each two-line data set contains the number of back seat passengers (a positive number less than 16) on the first line and the garbled sentence (containing at least one and at most 100 characters) on the second line.

The Output:

For each input set, you are to print the individual sentences from each person in the back seat. Then, print the number and the sentence of the person whose directions should be followed. Print a blank line between output for different data sets, and follow the format of the Sample Output below.

Sample Input:

```
2
Turn right here!!!!!Go left at the next light!!!!!!
3
Go !Are we !I'm hun!faster!!!!!!!there yet?!!!!!gry!!
```

Sample Output:

```
Person #1:  Turn right here!!
Person #2:  Go left at the next light!!!
Follow person #2:  Go left at the next light!!!

Person #1:  Go faster!!!
Person #2:  Are we there yet?!!
Person #3:  I'm hungry!
Follow person #1:  Go faster!!!
```

Where's the Bee?

Filename: BEEHIVE

A queen bee wants to monitor the construction of honeycomb sections in her beehive, and wants to keep track of which of the hexagonal cells in a given honeycomb section have been prepared as queen bee birthing cells. Each honeycomb section is arranged in columns of hexagonal cells. There are from 1 to 20 columns, and each column contains 1 to 9 cells. The cells are built downward from the top of the hive, and the cells in even columns start half a cell-height lower than the cells in odd columns, in order to pack the hexagonal cells as closely together as possible.

The Problem:

Given a description of a honeycomb section, draw a map of that section that indicates the position of any queen bee birthing cells.

The Input:

There will be several sets of data in the input, each representing a different honeycomb section.

The first line of each data set will contain several integers (separated by spaces), each indicating the number of cells in that column of the honeycomb section. If the line contains the numbers 3, 6, and 4, for instance, there are three columns--the first has 3 cells, the second has 6, and the third has 4.

The second line of the data set will contain exactly one integer *n*, indicating the number of queen bee birthing cells in the honeycomb section. The number of birthing cells will be less than or equal to the total number of cells (in the example above, $n \le 13$, since 3+6+4 = 13).

The next *n* lines of the data set will each contain a pair of integers (separated by a space) representing positions of the queen bee birthing cells. The first integer indicates the column, and the second is the cell of that column, counting from the top of the column.

The Output:

Print out the map of the honeycomb section, using the underscore (_), slash (/), backslash (\), and uppercase Q characters. Place the Q in the top portion of cells which are queen bee birthing cells. Each map should start in column 1, and there should be exactly one blank line between maps.

Sample Input:

```
1 1 1
2
1 1
3 1
3 4 2 3 2
4
2 2
2 3
3 2
4 2
```

Sample Output:

```
 _   _
/Q\_/Q\
\_/ \_/
  \_/
```

```
 _   _   _
/ \_/ \_/ \
\_/ \_/ \_/
/ \_/Q\_/ \
\_/Q\_/Q\_/
/ \_/ \_/
\_/Q\ / \
  \_/ \_/
  / \
  \_/
```

Vocal Punctuation

Filename: `VOCAL`

There's a comedian by the name of Victor Borge who is famous for, among other things, reading from a book and making noises to indicate punctuation. The best spelling I can come up with for the sounds he uses are as follows:

		pthut
!	:	phsss-pthut
,	:	pst
-	:	fshhh
'		chic
"	:	chic-chic
?	:	pssu-pthut

The Problem:

Victor would love it if you could write a program to read in several paragraphs of text and replace the punctuation with the phonetic spellings listed above.

The Input:

The input will consist of several paragraphs of text. There will be a maximum of 80 characters on any given line. Each paragraph will be separated by a blank line.

The Output:

The output from your program should be the input paragraphs with the punctuation replaced by the phonetic spellings inside square brackets. Do not make any other changes to the input.

Sample Input:

```
Due to the convergence of forces beyond his comprehension, Salvatore Quannucci
was suddenly squirted out of the universe like a watermelon seed, never to be
heard from again!

"I see", said the blind man, as he picked up his hammer and saw.
```

Sample Output:

```
Due to the convergence of forces beyond his comprehension[pst] Salvatore Quannuc
ci
was suddenly squirted out of the universe like a watermelon seed[pst] never to b
e
heard from again[phsss-pthut]

[chic-chic]I see[chic-chic][pst] said the blind man[pst] as he picked up his ham
mer and saw[pthut]
```

Dimwit Numbers

Filename: `DIMWIT`

On the planet Dimwit there lives a race of creatures called Dimwits. The Dimwits look basically like us, with two arms and two legs, but they have a variable number of fingers and toes on each limb. This causes some difficulty since they all grow up learning to count differently and thus use a numbering system that varies from Dimwit to Dimwit. Having no standard system causes them all sorts of confusion.

Each Dimwit is born with from 1 to 5 fingers or toes on each limb. A Dimwit that is born with 5 fingers on his left hand, 4 fingers on his right hand, 2 toes on his left foot, and 3 toes on his right foot, will use a numbering system called DW5423. This Dimwit will use his 3 right toes to count to 3, then use one toe on the left to count one group of four. Then, when all toes are used, he will use one finger on his right hand to represent one group of 12, and so forth.

All Dimwit numbers are 4 digits long, with leading zeroes if necessary. Dimwits use only non-negative integers; other types of numbers (such as reals) being far beyond the capacity of their distraught minds to handle. If a number is too large to represent with only 4 digits for a particular number of fingers and toes, then that number is simply called OVERFLOW by that particular Dimwit.

To illustrate how this variable radix number system works, here are some decimal conversions into a few of the possible Dimwit systems.

```
1   decimal  =  0001 DW5423  =  0001 DW1234  =  0001 DW1221
2   decimal  =  0002 DW5423  =  0002 DW1234  =  0010 DW1221
3   decimal  =  0003 DW5423  =  0003 DW1234  =  0011 DW1221
4   decimal  =  0010 DW5423  =  0004 DW1234  =  0020 DW1221
5   decimal  =  0011 DW5423  =  0010 DW1234  =  0021 DW1221
10  decimal  =  0022 DW5423  =  0020 DW1234  =  0120 DW1221
100 decimal  =  1310 DW5423  =  1200 DW1234  =  OVERFLOW DW1221
```

The Problem:

Help out the Dimwits by writing a program to convert numbers from any Dimwit number base to any other Dimwit number base.

The Input:

The first line will be a positive integer *n* indicating the number of conversions to be done. The next *n* lines will contain a 4 digit Dimwit number followed by one space, then the 6 character base of that number followed by one space, then the 6 character base to convert the number into.

The Output:

The output of your program should consist of *n* lines, each containing either the 4 digit number that has been converted, or OVERFLOW if the number is too large for the target base system.

Sample Input :

```
4
1310 DW5423 DW1234
1310 DW5423 DW1122
0001 DW1234 DW4321
1000 DW2222 DW3333
```

Sample Output:

```
1200
OVERFLOW
0001
0123
```

Snail

Filename: `SNAIL`

The Problem:

Given a two-dimensional matrix, your program is to print the matrix elements in a "snail-like" fashion as follows:

- print the elements in the top row from left to right
- print the elements in the rightmost column from top to bottom
- print the elements in the bottom row from right to left
- print the elements in the leftmost column from bottom to top
- print the elements in the second from top row from left to right
- . . .

Note that each element should be printed only once.

The Input:

The input is divided into sets. Each input set starts with two integer values for *m* and *n*, the dimensions of a matrix (both values are between 1 and 25, inclusive, and the two values are separated by exactly one space). The next *m* input lines contain *n* values for matrix rows, one row per input line. Assume that each matrix element is a single non-space character and that there is exactly one space between matrix elements in the input. End of data is indicated by end of file.

The Output:

For each matrix, print the elements in a "snail-like" fashion. Leave no space between elements. Leave one blank line after the output for each matrix.

Sample Input:

```
3 4
A B C D
J K L E
I H G F
5 6
A B C D E F
R S T U V G
Q Z Z Z W H
P Z Z Y X I
O N M L K J
```

Sample Output:

```
ABCDEFGHIJKL

ABCDEFGHIJKLMNOPQRSTUVWXYZZZZZ
```

In Line at the Bank

Filename: `BANK`

At Universal Bank there are five tellers. All customers wait in a single line for the tellers; when a teller finishes with a customer, the next person in line advances to that teller. Every customer will become irritated and leave if they have to wait in line for too long. At the beginning of the day, no teller is busy.

The following are the transactions that tellers can perform, along with the amount of time required for the teller to process each transaction:

Deposit	5 minutes
Withdrawal	7 minutes, plus 1 minute for each $100 or fraction thereof withdrawn
Get a Cashier's Check	10 minutes
Apply for a Loan	50 minutes, plus 3 minutes for each $100 or fraction thereof requested
Ask for Account Balance	2 minutes
Get an ATM Card	20 minutes
Open an Account	30 minutes

For example, it takes 9 minutes for a teller to process a withdrawal of $150 from an account (7 minutes + 2 minutes for a withdrawal amount more than $100 but less than or equal to $200).

The Problem:

Given a list of customers, their arrival times, their transactions, and the length of time they are willing to wait in line, produce a report detailing the departure times of each customer.

The Input:

There will be several data sets. The first line of each data set contains an integer *n* ($1 \leq n \leq 500$), the number of customers. The next *n* lines each contain a customer's *`name`*, *`arrival_time`*, *`wait_time`*, and *`transaction`*, with exactly one space between each of these fields. The customer's *`name`* is a string of at most 14 letters. The *`arrival_time`* is a positive integer indicating the number of minutes since the previous customer's *`arrival_time`*, or since the beginning of the day (time 0) for the first customer. The *`wait_time`* is a positive integer indicating the number of minutes that the customer will wait in line after arriving. The *`transaction`* is one of the following:

```
DEPOSIT               LOAN $amount          ATM_CARD
WITHDRAW $amount      ACCOUNT_BALANCE       OPEN_ACCOUNT
CASHIERS_CHECK
```

The transaction *`amount`* is a positive real number less than 10,000.

The Output:

For each data set, print a report with the following format (follow the Sample Output):

```
CUSTOMER        TRANSACTION/COMPLETED        IN        OUT
----------------------------------------------------------
name            transaction/YES         time_in   time_out
name            transaction/NO          time_in   time_out
...
```

The *name* and *transaction* fields should be echoed from the input file. The *time_in* and *time_out* values should be reported in minutes since the beginning of the day (time 0). If the customer successfully completes the transaction, then note YES next to the transaction; if the customer leaves before reaching a teller, note NO instead. Customers should be listed in order of arrival. Separate the output from different data sets by two blank lines. **NOTE:** Since arrival times in the input are relative, the "in" times and the "out" times in the output may exceed 32767.

Sample Input:

```
3
ROBERT 10 20 DEPOSIT
MIKE 20 20 WITHDRAW $1025.00
TRAVIS 20 20 CASHIERS_CHECK
6
GLENN 10 100 LOAN $500.00
MIKE 1 100 LOAN $500.00
LANCE 1 100 LOAN $500.00
JOEL 1 100 LOAN $500.00
ANA 1 100 LOAN $500.00
DAVID 1 30 OPEN_ACCOUNT
```

Sample Output:

```
CUSTOMER        TRANSACTION/COMPLETED        IN        OUT
----------------------------------------------------------
ROBERT          DEPOSIT/YES                  10         15
MIKE            WITHDRAW $1025.00/YES        30         48
TRAVIS          CASHIERS_CHECK/YES           50         60


CUSTOMER        TRANSACTION/COMPLETED        IN        OUT
----------------------------------------------------------
GLENN           LOAN $500.00/YES             10         75
MIKE            LOAN $500.00/YES             11         76
LANCE           LOAN $500.00/YES             12         77
JOEL            LOAN $500.00/YES             13         78
ANA             LOAN $500.00/YES             14         79
DAVID           OPEN_ACCOUNT/NO              15         45
```

Base Alphabets

Filename: `BASE`

Two words are said to use the same **base alphabet** if they both use all of the same letters, although not necessarily using equal numbers of them. For example,

> **rescue** and **curse** use the same base alphabet (c, e, r, s, u)
> **rose** and **sore** use the same base alphabet (e, o, r, s)
> **ail** and **snail** do not use the same base alphabet
> **base** and **alphabet** do not use the same base alphabet

The Problem:

Given two words, determine whether they use the same base alphabet.

The Input:

The input file will contain pairs of words, one word per line. Each word will consist of only lower case letters, will contain at least one letter, and will contain at most 30 letters.

The Output:

Print each word exactly as it appears in the input file, one per line. After each pair of words, print one of the following messages:

```
The words share the base alphabet (x, x, ..., x).
The words do not share the same base alphabet.
```

whichever is appropriate. The *x*'s should be replaced by the lower case letters in the shared base alphabet; the order of the letters is not important. Print a blank line after each message.

Sample Input:

```
rescue
curse
base
alphabet
```

Sample Output:

```
rescue
curse
The words share the base alphabet (c, e, r, s, u).

base
alphabet
The words do not share the same base alphabet.
```

Nothin' But Net

Filename: `NOTHIN`

Michael recently accepted a position as an ad executive. His first few commercials starred famous basketball players, who made trick shots to promote a fast food restaurant chain. These were extremely popular, but he is quickly running out of ideas and needs your help.

The Problem:

Write a program that creates new situations for fast food restaurant commercials.

The Input:

There will be multiple data sets.

The first line of each data set will contain a number, *n*, for the number of objects that are involved in the commercial.

Each of the following *n*-1 lines will contain a string of letters and possibly spaces, where the *i*-th string is the name of object *i*. The *n*-th object will always be a basketball net.

On the next *n*-1 lines will be an *n-1* by *n* matrix of integers from 0 to 3 representing how the objects are related in basketball terms. If there is a non-zero integer at row *i* and column *j* in the matrix, a basketball can travel from object *i* to object *j*. If the integer is a 1, then the basketball will bounce "`off the`" object *i* to reach object *j*. If the integer is a 2, the basketball will go "`through the`" object *i* to reach object *j*. If that number is a 3, the basketball will go "`around`" object *i* to reach object *j*. Once the basketball has reached the net, it will stop.

The matrix will always be set up so that it is not possible for the basketball to continuously travel between objects (for example, the basketball will not be able to go from object 1 to object 2 to object 3 and then back to object 1).

After the matrix there will be a list of integers indicating objects which the basketball players initially shoot at (the integer *i* indicates object *i*). There will be one integer per line, and the end of the list will be indicated by the integer 0.

The Output:

For each starting object in the list, print a sentence describing a path that the ball could take from the starting object to the basketball net, ending with the phrase, "`nothin' but net.`" Capitalize the first word of each output sentence, but other than that, preserve the capitalization of the object names. Follow the format of the Sample Output, and print a blank line between output sentences. For any given starting object, there may be multiple possible paths; your program should only print one path.

Sample Input:

```
6
Saturn
skyscraper
highway
window
scoreboard
0 3 0 0 0 0
0 0 1 0 0 1
0 0 0 1 0 0
0 0 0 0 2 0
0 0 0 0 0 1
4
3
2
1
0
```

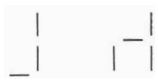
Sample Output:

```
Through the window, off the scoreboard, nothin' but net.

Off the highway, through the window, off the scoreboard, nothin' but net.

Off the skyscraper, off the highway, through the window, off the scoreboard, not
hin' but net.

Around Saturn, off the skyscraper, off the highway, through the window, off the
scoreboard, nothin' but net.
```

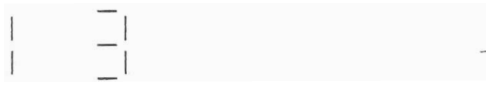
Visual Acuity

Filename: `ACUITY`

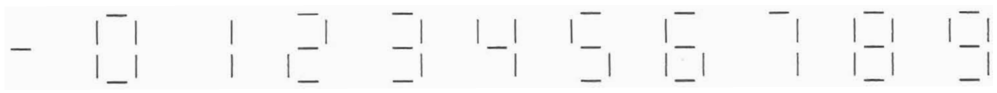
Fred has a vision disorder which causes him to perceive digital displays as if they had been flipped along a horizontal line through the center of the numbers. For example, he once complained that he could not read the number 74 on a digital display, because he saw the following:



However, he had no problem reading the number 13, since he saw the following:



Thus, in some cases, Fred will see an invalid number, but in others he will see a valid number. However, there are situations in which the valid number that Fred sees is not the same as the number on the digital display. For example, he perceives the number 22 as 55. **Note:** Refer to the table below for the representations of digits and the negative sign on the display for the purposes of this problem.



The Problem:

Given a digital readout, determine whether Fred sees a valid number. If so, determine whether the number Fred sees is bigger than the number actually on the display.

The Input:

Each line of the input file will contain exactly one integer *n*. This integer may be up to twenty characters long and may be negative.

The Output:

For each input line, print one of the following messages (whichever applies):

```
Fred does not see a valid number.
Fred sees x, which is not bigger than n.
Fred sees x, which is bigger than n.
```

Sample Input:

```
74
13
22
```

Sample Output:

```
Fred does not see a valid number.
Fred sees 13, which is not bigger than 13.
Fred sees 55, which is bigger than 22.
```