

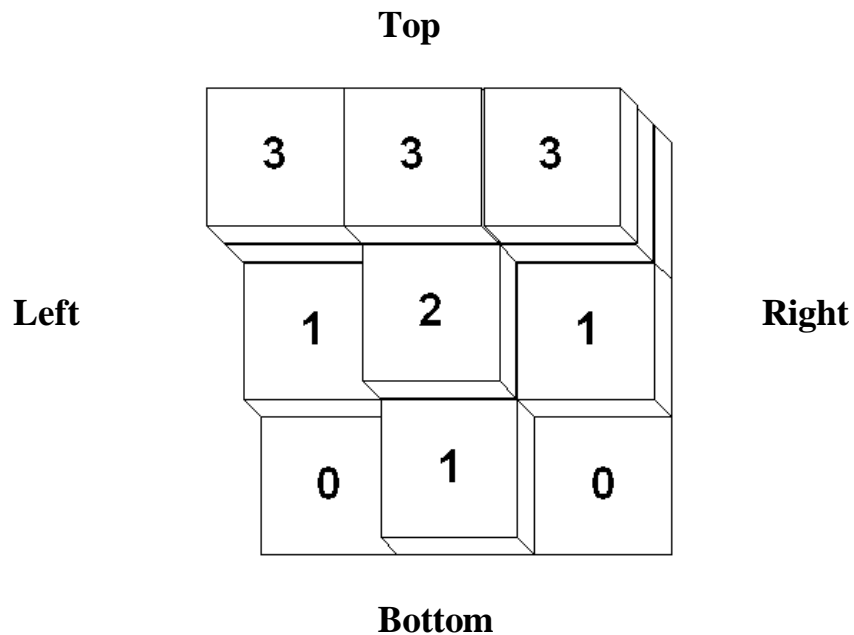
ACM South Central Region 1999 Programming Contest

Problem #1: Lego Land

Source file: lego.ext

Input File: lego.dat

Output File: lego.out



Introduction

As an urban Lego engineer, it is vital to obtain every possible technological advantage over your competition. You have recently noticed how popular virtual reality technology is with human engineers and wish to have something similar to help woo venture capitalists. Luckily, a VR view of a Lego downtown is very simple. First, you only have to show buildings, and you already know the heights and locations of all the buildings you want to build. Secondly, Lego VR is only capable of presenting a view of cube-shaped section of the city. It does this by projecting the cube onto one of the planes containing one of the cube's faces. Lego engineers may browse the view by rotating the cube-shaped city section 90 degrees at a time about any one of the X, Y, or Z-axes passing through its center.

You are to write a Lego VR program that will take as input an $N \times N$ matrix view of a downtown area and a series of rotational commands and produce the resulting $N \times N$ matrix view of the same downtown area from the resulting perspective. The $N \times N$ input matrix will represent an overhead view of an $N \times N \times N$ cube of a Lego downtown (with one face of the cube at ground level).

Each number in the input matrix indicates the height of the building in the corresponding location. For example, 0 indicates that no building is present, while 3 indicates a building 3 stories (Legos) high. Note, in the initial overhead view there is no hidden open space. In other words, the buildings are solid. Also note that only buildings are represented in the view, not the ground. Any region of the view that does not contain a block of a building will have value 0.

Input

Input consists of an unspecified number (at least one) of sets of three sections:

1. Grid size – A single line containing a single integer N ($1 \leq N \leq 9$) which represents the size of the cube of buildings represented.
2. Building heights – An $N \times N$ array of whole numbers in the range 0 to N (inclusive). Each row in the array is given on its own input line, and array elements are **not** delimited. This initial $N \times N$ array represents an overhead view of an $N \times N$ **solid** structure of Lego buildings, with each number representing the height of the building in the corresponding location. Height of 0 indicates that no Lego exists in the given location.
3. A series of rotational commands about the center of the $N \times N \times N$ cube containing the Legos, one of:
 - “X” – rotates about the horizontal axis in the current view. The cube rotates such that the face of the cube that is at the **top** of the current view moves forward to become the new viewing face.
 - “Y” – rotates about the vertical axis in the current view. The cube rotates such that the face of the cube that is at the **left** of the current view moves forward to become the new viewing face.
 - “Z” – rotates about the line-of-sight axis in the current view. The viewing face does not change but is rotated 90 degrees counterclockwise.

These commands are a single-unbroken string with at most 80 commands

Between each two sets of input data is exactly one blank line. There will be no white space before the first input set or following the last input set.

No invalid input will be provided.

Output

For each input case, output consists of an $N \times N$ array of numbers in the range 0 to N (inclusive) representing the final view for each input array and its associated set of rotations. Output arrays must be separated by exactly one blank line, and there must be no white space following the last digit of the last output array. Similarly, the first digit of the first output array must be the first character in the output file.

Note: No output to the screen should be provided, and no extraneous output of any kind should be sent to the specified output file.

Since this program essentially generates views of 3-dimensional objects, it is good to be very specific in describing how these views are calculated. Looking at the figure above, note that the cube has six sides, top, left, bottom, and right are labeled. The front is the side of the cube nearest the viewer, and the back is the side farthest from the viewer. The initial input view is meant to represent a Lego city from above. Each number represents the heights of the building at that location (or correspondingly, the maximum distance from the back of the cube for any Lego piece in that row and column). When the cube is rotated, the numbers no longer indicate the height of buildings since the view is no longer (necessarily) a view from above. Instead, the numbers continue to represent the maximum distance from the back of the cube for any Lego piece in that row and column of the cube. The idea is that the numbers will represent those Legos that are nearest the viewer, and therefore those Legos that are visible to the viewer.

Sample Input

```
3
010
021
010
Z

5
11111
22222
```

33333
44444
55555
XXX

9
000010000
000020000
000030000
000040000
135797531
000040000
000030000
000020000
000010000
ZXXXXZYZZZZZXXXXZYZZZZZ

Sample Output

010
121
000

00005
00055
00555
05555
55555

500000000
550000000
555000000
555500000
988776655
555500000
555000000
550000000
500000000

ACM South Central Region 1999 Programming Contest

Problem #2: Shotgun!

Source file: shotgun.ext

Input File: shotgun.dat

Output File: shotgun.out

Introduction

Everyday when the FORMRunner team goes to lunch there are several issues: Who is driving? Who gets shotgun? How many cars do we need to take? Where are we going?

Once we determine who is driving, who is shotgun, and other seating preferences, we face the problem of getting everyone in the cars in a way that meets everyone's preferences.

You have 2 cars that can each hold 4 people. There are 8 people. You will be given a set of conditions such as:

- "Bob will not ride with Sue"
- "Sally is a driver"

The possible conditions are:

- X will not ride with Y
- X and Y are in the same car
- X is a driver
- X is a passenger
- X is shotgun

Since the cars (and thus, the drivers) have already been chosen, it is safe to assume that no two drivers are requesting to travel in the same car. However, the other seating arrangements are subject to the individual biases of the group, and therefore may or may not be satisfiable.

Here are some points to consider:

Every person will be either a driver, shotgun, or passenger.

All drivers and shotguns will be explicitly stated, though passengers may or may not be stated.

Each car will have exactly one driver, one shotgun, and two passengers.

A shotgun may sit at the shotgun position of any available car.

No condition of the type "X and Y are in the same car" will appear where both X and Y are drivers. FORMRunner people at least have that much common sense.

Also, the contradictory conditions that "X will not ride with Y" and "X and Y are in the same car" (as well as other contradictory variations on these two types of conditions) will never be presented in the same input set.

At most, one condition of the type "X is [a] Z" will appear for each person (X) in any input section.

Input

The input file will consist of an unspecified number of sections fitting this description:

A single line containing 8 names (each no more than 10 alphabetical characters) delimited by single spaces: "Bill Bob Buddy ..."

set of conditions: one on each line

A blank line or end of file indicates the end of the condition list for this section.

Each input section will be separated from the others by exactly one blank line, and there will be no blank lines following the last condition of the last section or preceding the first line of the first section.

No invalid input will be provided.

Output

Print out an answer for each section, to the question: Is a seating arrangement possible?

Yes

-or-

No

Note: No output to the screen should be provided, and no extraneous output of any kind should be sent to the specified output file.

Sample Input

Mary Bob Sue Joe Jane Chris Lisa Mike
Mary is a driver
Bob is a driver
Sue is shotgun
Mike is shotgun
Sue and Mike are in the same car

Mary Bob Sue Joe Jane Chris Lisa Mike
Sue will not ride with Chris
Chris is a driver
Mike is shotgun
Mary is a driver
Sue is shotgun

Mary Bob Sue Joe Jane Chris Lisa Mike
Mary is a driver
Joe and Chris are in the same car
Bob is a driver
Mary will not ride with Jane
Joe is shotgun
Mike and Lisa are in the same car
Jane is shotgun
Mike will not ride with Jane

Sample Output

No

Yes

No

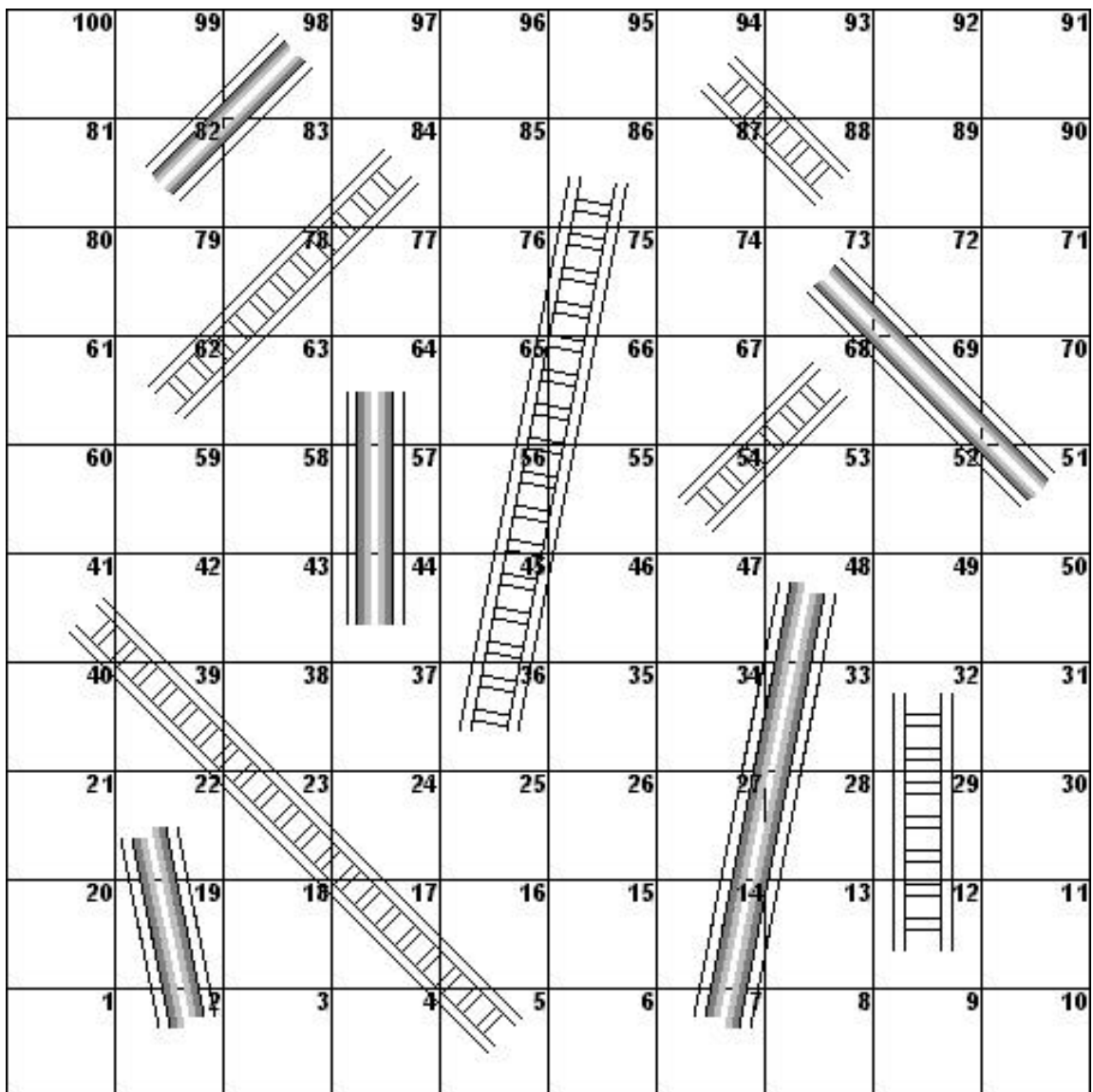
ACM South Central Region 1999 Programming Contest

Problem #3: Child's Play

Source file: child.ext
Input File: child.dat
Output File: child.out

Introduction

In this problem, you will simulate the child's game "Chutes-n-Ladders". In Chutes-n-Ladders, a player starts at position 1 on the board and tries to reach the position 100. Play consists of a series of moves each of which starts with the player rolling a pair of dice. The player advances on the board by the number of positions rolled. If any move ends on a board position that contains the top of a Chute, he "slides down" the chute to the position at the end of the Chute. For example, a move that ends on the board position 73 below causes the player to slide down to board position 51. If a move ends on a position containing the bottom of a ladder, the player "climbs" the ladder to the board position at the top of the ladder. For example, a move that ends on board position 62 below causes the player to climb up to board position 84.



Your program should use the board above in this simulation. For absolute clarity, Table 1 and Table 2 below are exhaustive lists of the chutes and ladders on the above board.

Chute Number	Starting Board Position	Ending Board Position
1	22	2
2	48	7
3	64	44
4	73	51
5	98	82

Table 1 : Enumerated List of Chutes

Ladder Number	Starting Board Position	Ending Board Position
1	5	41
2	12	32
3	36	86
4	54	68
5	62	84
6	88	94

Table 2 : Enumerated List of Ladders

Some other rules your program should be designed to implement:

1. Every game starts with the player at position 1.
2. Each roll of the die advances the player's position incrementally up the board up by the number indicated by the die. If the player ends the advance on a board position that contains the top of a chute or the bottom of the ladder, the player is moved to the end of the chute or ladder.
3. If the player advances to position 100, the player "wins" the game. Note that any role of the die that would advance the player beyond position 100 simply advances to position 100 and stops with a win.

Input

Input to your program consists of a series of "games" each on a single line. Each game consists of the name of the player left justified in columns 1-10. A series of 20 rolls of a pair of die ($2 \leq \text{roll} \leq 12$) then follows beginning column 12. Rolls are separated by single space and there are no leading zeroes. There are no trailing spaces after the 20th integer roll. For each input case/game, your program will run a simulation of a "Chutes-n-Ladders" game for a single player. Your program is to use the input die rolls to determine the final position of the player after 20 rolls. If at any point in the simulation the player reaches board position 100, your program will declare the player a winner.

No invalid input will be provided.

Output

For each input case, your program will produce exactly 1 line of output. Columns 1-10 must contain the player's name left justified appearing exactly as in the input stream. Column 12 starts the description of the result of the simulation. If after running the simulation for all 20 rolls of the die on the input line the player has not reached board position 100, your program should print "position X" where X is the final position of the player after the 20th roll. X should be left justified beginning in column 21 and should not have leading zeroes. If at any time in the simulation a player reaches board position 100, your program should print "wins" in columns 12-15. There should be no other output or blank lines produced by your program.

Note: No output to the screen should be provided, and no extraneous output of any kind should be sent to the specified output file.

Sample Input

Buckwheat 7 9 12 4 5 9 8 3 6 11 7 5 2 10 9 6 7 3 5 10

Darla 4 10 11 4 6 9 8 5 10 11 2 3 8 7 11 9 4 3 12 12
Weaser 4 7 8 7 3 10 3 5 8 5 4 2 3 4 12 3 3 4 12 5

Sample Output

Buckwheat wins
Darla wins
Weaser position 99

ACM South Central Region 1999 Programming Contest

Problem #4: Soundex

Source file: soundex.ext

Input File: soundex.dat

Output File: soundex.out

Introduction

You have been given two lists of names from a small community. One list comprises people who currently live in the town; the other is a list of people whom have passed away leaving inheritances with no known living relatives. The Town council wishes to apportion this wealth to the closest living relatives in the town. Unfortunately the town records have been lost in a fire. The councilmen have decided to determine possible relatives by comparing names by their sound roots (i.e. soundex). Given the two lists, match possible relatives to the deceased.

Soundex Coding Guide

To compute the soundex value of a person's name, your program will use the first letter (even if it is one of the ignored letters in the soundex value table) of the person's surname, followed by 3 digits taken from the following table for subsequent letters in the person's surname. If the surname contains an insufficient quantity of letters to complete the soundex value, your program will simply pad the soundex value with zeroes to complete the 3 digits. For example, Mr. T's surname would simply be coded as T-000; and Mike Mart's surname would be coded as M-620. Soundex values are limited to a maximum of 3 digits after the first letter regardless of how long the surname may be.

Soundex Value	Mapped Letters
1	B, F, P, V
2	C, G, J, K, Q, S, X, Z
3	D, T
4	L
5	M, N
6	R
Ignored	A, E, I, O, U, H, W, Y

Additional Soundex Coding Rules

1. Names With Double Letters

If the surname has any double letters (even doubles that include the first letter of the surname), they should be treated as one letter.

Gutierrez is coded G-362 (G, 3 for the T, 6 for the first R, second R ignored, 2 for the Z).

2. Names with Letters Side-by-Side that have the Same Soundex Code Number

If the surname has different letters side-by-side with the same soundex value, they should be treated as one letter.

Pfister is coded as P-236 (P, F ignored, 2 for the S, 3 for the T, 6 for the R).

Jackson is coded as J-250 (J, 2 for the C, K ignored, S ignored, 5 for the N, 0 added).

3. Case sensitivity

Your program should be completely case insensitive when computing soundex values.

Input

Input to your program consists of a list of deceased townspeople and a list of current townspeople. The first line of input consists of a single left-justified integer ($1 \leq D \leq 100$) with no leading zeroes indicating the number of deceased townspeople. Lines 2 through $D+1$ each contain a single surname of the deceased. Line $D+2$ contains a single left-justified integer ($1 \leq T \leq 100$) with no leading zeroes indicating the number of current townspeople. The next T lines each contain a single surname of a current townspeople.

A surname in either list consists of up to 20 alphabetic characters. All surnames are left justified and contain only mixed-case alphabetic characters. There are no leading, trailing, or embedded non-alphabetic characters on any line containing a surname.

No invalid input will be provided.

Output

Output from your program consists of a single line of output for each deceased townspeople. Each output line consists of a name of a deceased townspeople followed immediately by a colon followed by the list of current townspeople whose soundex values are the same as the deceased person. Names on the list of current townspeople on the output line should be preceded by a single space. If there are no current townspeople whose soundex value matches the deceased townspeople, your program should print “(none)” after the colon. Two examples of exact required formatting (soundex values aside) are as follows:

```
Firstdeceased: Firstcurrent Secondcurrent Thirdcurrent  
Seconddeceased: (none)
```

Your program should retain the case of the input surnames for use on the output file.

Note: No output to the screen should be provided, and no extraneous output of any kind should be sent to the specified output file.

Sample Input

```
4  
McKeger  
Rodriguez  
Navalsha  
Miller  
7  
Johnson  
Alley  
Moliere  
MacJoy  
Nebalsh  
Neopolik  
Rothworski
```

Sample Output

```
McKeger: (none)  
Rodriguez: Rothworski  
Navalsha: Nebalsh Neopolik  
Miller: Moliere
```

ACM South Central Region 1999 Programming Contest

Problem #5: Drive Safely

Source file: drive.ext
Input File: drive.dat
Output File: drive.out

Introduction

As the host one of this year's many millenium keg parties, you feel that it is your responsibility to ensure that none of your driving patrons leave the party in a legally intoxicated state. Being an industrious and scholarly individual, you've decided to use a method based on scientific evidence to keep them off the street. Every driver entering the party must check their keys at the door and be weighed. At the bar, every drink imbibed by a driver is recorded, along with the time of the drink. When a driver then decides to leave, you quickly calculate his current blood alcohol content (BAC), considering his metabolism as indexed by his weight and the times and concentration of his drinks, to determine if it is below the legal limit of 0.10. If he is below the limit, he may leave. Otherwise, he must return to the party and may try again to leave at a later time.

Here are some facts that you have discovered during your research into blood alcohol levels:

The effect of an ounce of drink on a person's BAC is inversely proportional to the individual's weight according to the following equation:

$$\langle \text{BAC increase} \rangle = \langle \text{proof} \rangle / \text{weight}$$

A person's BAC at the time of a drink is calculated by summing the BAC at that time due to previous drinks and the BAC increase provided by the current drink

A person's blood alcohol content $\langle n \rangle$ minutes after his last drink is calculated as:

$$\langle \text{BAC at time of last drink} \rangle - ((\langle \text{metabolism} \rangle * \langle n \rangle) / 60)$$

If the previous equation evaluates to a value less than zero, the person's BAC is 0
BAC of 1 is not a limit. BAC is merely a scalar that must be non-negative.

Assume the following:

At most 50 total drivers will attend the party.

Everyone is a driver.

Drinks are imbibed completely and instantaneously

A person's BAC increases instantaneously at the time of the drink

All drinks are precisely one ounce each.

A person's weight is an integer number and remains constant during the party.

A person's metabolism relates to that person's weight according to the following chart:

BAC is not an integer, yet two digits of precision shall be enough to unambiguously determine whether or not a person is capable of leaving the party.

No two party members will have the same name.

Every time a party member arrives their BAC is assumed to be zero.

The weight is always indicated when the party member arrives.

Weight Range	Metabolism
0 - 50	0.25

51 – 100	0.5
101 – 150	0.75
151 – 200	1.00
201 – 300	1.50
301 and up	2.00

Input

Three sections:

1. List of drinks – consists of:
 - Number of drinks – a single integer x , where $0 < x < 20$, indicating the number of types of drinks. This is the only entry on this line.
 - x Drink Entries – each of which consists of a non-empty string of at most 20 alphanumeric characters with no embedded spaces (a drink name), followed by a space and an integer p , where $0 \leq p \leq 200$, indicating the proof of the drink. Each entry is on a separate line.
2. Blank line
3. List of events – consists of less than 100 events and will not be empty. Each event is on a separate line and has one of the following formats:
 - (a) `<time> <person> arrives <weight>`
 - (b) `<time> <person> drinks <drink>`
 - (c) `<time> <person> asksforkeys`

`<time>` is an integer t , where $0 \leq t \leq 10000$ and indicates the number of seconds since the start of the party that a given event is occurring. `<time>` for any given event will be greater than `<time>` for the previous event.

`<weight>` is an integer w , where $0 < w < 1000$ and indicates the weight of `<person>`.

All instances of `<person>` will be non-empty strings of at most 20 alphanumeric characters with no embedded spaces. All parameters of an event will be separated from each other by exactly one space, and there will be no trailing spaces

Events will only appear in a sensible order. For instance, a person will not have a drink or ask for keys unless that person has arrived and has not since left.

No invalid input will be provided.

Output

There is only one set of input. For each event of type (c) in that set, one of the following lines must be written to the output file:

- (a) `<time> <person> leaves`
- (b) `<time> <person> stays`

`<time>` is the same in either case, and is the `<time>` value from the corresponding event. `<time>` values in the output file must be in ascending order. Line (a) is written to the output file if and only if `<person>` has a BAC strictly less than 0.10 at `<time>`. Otherwise, line (b) must appear in the output file. Formatting of output is similar to formatting of input. All parameters are separated by exactly one space, and no line may contain trailing spaces.

Note: No output to the screen should be provided, and no extraneous output of any kind should be sent to the specified output file.

Sample Input

2
gin 100
margarita 50

0 Dave arrives 100
3 Hershey arrives 1000
5 Dave drinks gin
10 Dave asksforkeys
500 Dave asksforkeys
1000 Hershey drinks margarita
1002 Hershey drinks margarita
1003 Hershey asksforkeys

Sample Output

10 Dave stays
500 Dave leaves
1003 Hershey leaves

ACM South Central Region 1999 Programming Contest

Problem #6: Where Am I?

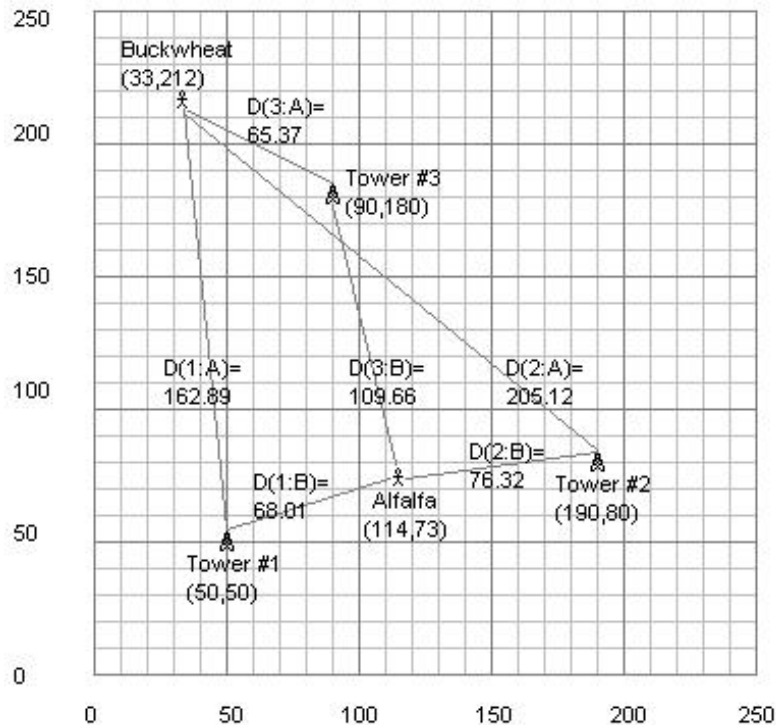
Source file: where.ext

Input File: where.dat

Output File: where.out

Introduction

Your circle of friends has created their very own GPS receiver. You are responsible for the embedded software component that computes the position of the GPS on a first quadrant grid when given the distance to each of the three fixed position broadcast towers. In the figure below, two example positions are shown. For Buckwheat's example position, your program is given the distances between Buckwheat and the three towers. Your program will use the positions of the three towers at (50,50), (190,80), and (90,180), and the distances to the towers of 162.89, 205.12, and 65.37 respectively to compute the position of Buckwheat of (33,212) when rounded to the nearest integer coordinates. Likewise, your program will use the distances of 68.01, 76.32, and 109.66 to compute the position of Alfalfa as (114,73).



Input

Input to your program will consist of an indeterminate number of test cases. Each test case will be contained on a single line and consists of the name of the person holding the GPS (alphabetic characters in columns 1-10 with trailing spaces as required) and three distances from the GPS position to the three towers. All distances are represented as floating point numbers with exactly two decimals of precision. The first distance is the distance to tower #1 (at fixed position 50,50); the second distance is the distance to

tower #2 (at position 190,80); and the third distance is the distance to tower #3 (at position 90,180). You are guaranteed that 1) all three distances unambiguously converge near a single point and 2) all points fall within quadrant I with x and y values between 0.00 and 250.00. Note that since the distances are rounded to 2 decimal places, they will only converge near as single point and not mathematically on a single point. Your program will have to consider this limitation.

You may assume that there are no invalid or improperly formatted lines of input. Your program should continue reading input until End-of-File.

No invalid input will be provided.

Output

For each input case, your program should print exactly one line of output containing the name of the person holding the GPS for that test case, in columns 1-10, followed by the GPS's integer coordinates of the form (x,y) starting in column 12. Your program should compute the (x,y) coordinates accurate to the nearest integer for each dimension. Also note that there are no leading zeroes nor embedded blanks in the printed GPS coordinates from you program.

There must be no empty lines or other extraneous output in the output file.

Note: No output to the screen should be provided, and no extraneous output of any kind should be sent to the specified output file.

Sample Input

```
Buckwheat 162.89 205.12 65.37
Alfalfa   68.01 76.32 109.66
Spanky    238.30 142.58 133.15
```

Sample Output

```
Buckwheat (33,212)
Alfalfa   (114,73)
Spanky    (217,220)
```