UCF “Practice” Local Contest — Aug 24, 2013

The Suffix Game

filename: `suffixgame`

Dr. Orooji’s twins, Mack and Zack, have discovered a new game to help them pass the time. First, Mack will select a word from the dictionary and write it down. Next, Zack selects a word and writes it down below Mack’s word, taking care that the ends of the words are aligned. For instance, if Mack chooses the word "random", and Zack chooses "kingdom", then they will write them down as:

```
 random
kingdom
```

Observe that the shorter word has been padded with a space at the front, to make the ends of the words line up.

Next, they will take turns examining the last letter of each word. If both strings end with the same letter, it will be removed from the ends of both strings. This procedure continues until the final characters do not match. So, for the previous example, the procedure will give:

```
 random  -->   rando  -->   rand  -->   ran
kingdom       kingdo       kingd       king
```

There is a catch, though. When the game ends, the two resulting words are written down in a special notebook. Since the twins insist on writing *exactly* two words, neither word must be completely consumed. Thus, the shortening procedure will end if either of the strings is shrunk down to a single character (see Sample Input/Output for clarification).

The Problem:

Dr. Orooji has decided to beat the twins at their own game, literally. Your job is to assist him by writing a program that, given the two starting words of a game, will play the game to the end and output the two final words.

The Input:

Mack and Zack play multiple games. The first line of the input to your program will be an integer *g*, indicating the number of games to be played. This will be followed by *g* lines, with two words on each line, indicating the two initial words. The words will consist of only lowercase letters (`'a' - 'z'`), each word at least one letter and at most 20 letters, first word starting in column one and the second word separated from the first word by exactly one space.

The Output:

At the beginning of each test case, output "*Game #g:*", where *g* is the game number (starting from 1). Then, output the two input words (indented three columns). Finally, output the results of the game (indented three columns) by saying: "The words entered in the notebook are *w1* and *w2*.", where *w1* and *w2* represent the two final words, in the same order as the originals.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
4
random kingdom
win twin
crushed sofa
myucf ucf
```

Sample Output:

```
Game #1:
   The input words are random and kingdom.
   The words entered in the notebook are ran and king.

Game #2:
   The input words are win and twin.
   The words entered in the notebook are w and tw.

Game #3:
   The input words are crushed and sofa.
   The words entered in the notebook are crushed and sofa.

Game #4:
   The input words are myucf and ucf.
   The words entered in the notebook are myu and u.
```

UCF "Practice" Local Contest — Aug 24, 2013

Counting Triangles

filename: `counttri`

Triangle Tom is at it again. This time, however, instead of attempting to calculate areas of triangles, he just wants to count how many of them there are. Given a list of possible points, your job will be to help Tom figure out how many different triangles can be formed with those points.

The Problem:

Given a list of points in the Cartesian plane, determine how many triangles (with all angles strictly less than 180º) can be formed with those points.

The Input:

The first line of the file will contain a single positive integer n ($1 \le n \le 50$), denoting the number of test cases in the file. The first line of each test case contains one integer, k ($1 \le k \le 100$), denoting the number of points for that test case. The second line of each test case contains k ordered pairs of integers (separated by spaces) denoting the x and y coordinates of each point, respectively. It is guaranteed that $-100 \le x \le 100$ and $-100 \le y \le 100$ for all coordinates. It is also guaranteed that each point given will be unique.

The Output:

At the beginning of each test case, output "`Test case #`*`t`*`:`", where *t* is the test case number (starting from 1). Follow this with a statement of the following form:

x `triangle(s) can be formed.`

where *x* represents the distinct number of triangles that can be formed with the given points.

Leave a blank line after the output for each data set. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
2
4
1 1 2 2 3 4 4 4
3
2 4 6 8 7 9
```

Sample Output:

```
Test case #1: 3 triangle(s) can be formed.

Test case #2: 0 triangle(s) can be formed.
```

UCF "Practice" Local Contest — Aug 24, 2013

UCF Hold-em Poker

filename: `holdem`

Dr. Orooji's children, as if they don't spend/waste enough time on Nintendo, want to learn some card games. So, they asked Dr. O to teach them poker and Texas Hold-em poker. Since poker (and the Texas Hold-em version of it) is rather complicated, Dr. O decided to teach a simplified version of it. And, since you are programming for free today, why not use you guys to help Dr. O!

A poker hand consists of five cards. The possible hands we will consider for this problem, from worst to best, are as follows:

BUST: any five cards that do not form one of the following hands

TWO OF A KIND: any two cards which match in value plus three other cards

THREE OF A KIND: any three cards which match in value plus two other cards

FULL HOUSE: any THREE OF A KIND plus TWO OF A KIND of a different value

FOUR OF A KIND: any four cards which match in value plus one other card

In our simplified version of Texas Hold-em (called UCF Hold-em), you are given seven cards and want to determine the best hand among the five possibilities above. That is, if you were to choose five of the seven cards, what is the best poker hand you can make?

The Problem:

Given seven cards (chosen from a single valid playing deck of cards), you are to determine the best poker hand using five of the cards. You'll choose the cards that will result in the best possible hand from the above five possibilities.

The Input:

There will be multiple test cases. The first input line contains a positive integer *n*, indicating the number of data sets to be processed. The data sets will be on the following *n* input lines, each set on a separate line. Each set contains seven characters, starting in column one (with no spaces anywhere). The characters will be from the string "`23456789TJQKA`", representing the 13 possible card values.

The Output:

At the beginning of each test case, output "`UCF Hold-em #`*h*`:`", where *h* is the hand number (starting from 1). Then, output the input hand, followed by the best possible hand (on the next line). Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
5
2645372
83A5A2A
29T9TT3
JQ3Q4QQ
8756324
```

Sample Output:

```
UCF Hold-em #1: 2645372
Best possible hand: TWO OF A KIND

UCF Hold-em #2: 83A5A2A
Best possible hand: THREE OF A KIND

UCF Hold-em #3: 29T9TT3
Best possible hand: FULL HOUSE

UCF Hold-em #4: JQ3Q4QQ
Best possible hand: FOUR OF A KIND

UCF Hold-em #5: 8756324
Best possible hand: BUST
```

UCF "Practice" Local Contest — Aug 24, 2013

Cell Phone Contacts

filename: `contacts`

Modern cellular phones have a small chip, called a SIM card, that allows the phone to interface with the carrier's network. SIM cards also have a small amount of memory onboard where the user can store data, such as names, phone numbers, and e-mail addresses. A common problem with SIM card memory is that it cannot group several numbers under the same name, as most cell phones can. This poses a problem when the user wishes to switch phones. The easiest way to transfer contacts is to copy them to the SIM card from the old phone, and then copy them back from the SIM card to the new phone. However, if the user has multiple phone numbers and/or e-mail addresses for a given name, the contact information will be broken up, with multiple entries of the same name, each associated with a different phone number or e-mail address.

Obviously, the user of the new phone would like for his or her contact data to be organized, with each contact's name listed only once, and the various phone numbers and e-mail addresses for that contact organized under that single entry. You are to write a program to help with this.

The Problem:

Given the contact data from a SIM card, consisting of pairs of names and email addresses or names and phone numbers, generate organized input data for the new cell phone. The new cell phone has a strict data format that must be followed precisely (the format is described in The Output section below).

The Input:

There will be multiple contact lists (test cases) to process. The first line of each contact list contains a single integer, n ($1 \le n \le 100$), indicating the number of contact data entries on the SIM card. On each of the next n lines will be a name, followed by either a phone number or an e-mail address. Names consist of exactly two strings of upper- and lower-case alphabetic characters (the first name followed by the last name). Each first (last) name is at least one character and at most 20 characters. There will be exactly one space between the first and last names in input, and one space between the last name and the phone number or e-mail address. A phone number will consist of exactly 10 digits (no hyphens). An e-mail address will consist of a string of letters and/or numbers separated by exactly one at ('@') sign, and at least one dot ('.'). No input line will exceed 80 characters. There will be no leading or trailing whitespace on any line.

End of input is indicated by a value of 0 for n. This case should not be processed.

The Output:

For each data set, print the heading "`Contact list #`*`d`*`:`" where *d* is the set number in the input (starting with 1). Then, for each unique contact (by name) in the set, output a contact entry. Note that case is significant, so "John DOE" is a different contact from "John Doe". Contact entries must be output in the following format:

<Contact Name>
Phone:
<Phone Number 1>
<Phone Number 2>
…
<Phone Number p>
E-Mail:
<E-Mail Address 1>
<E-Mail Address 2>
…
<E-Mail Address q>
###

Phone numbers should be printed in the format "`(123)456-7890`". E-mail addresses should be printed exactly as input. Contact entries should be output in ascending ASCII order by last name, then first name. Within each contact, phone numbers should be output in numerical order, and e-mail addresses should be output in ascending ASCII order. There must be no leading or trailing spaces in the output. (Note that the built-in string comparison functions do compare strings in ASCII order.)

Leave a blank line after the output for each data set. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
6
John Doe 4071234567
Bill Smith bsmith@somewhere.com
Bill Smith 1231231234
John Doe John.Doe@my.house.net
John Doe John.Doe@work.com
Bill Smith 1234567890
2
Bone Head 1231231234
Air Head airhead@my.house.net
0
```

Sample Output:

```
Contact list #1:
John Doe
Phone:
(407)123-4567
E-Mail:
John.Doe@my.house.net
John.Doe@work.com
###
Bill Smith
Phone:
(123)123-1234
(123)456-7890
E-Mail:
bsmith@somewhere.com
###

Contact list #2:
Air Head
Phone:
E-Mail:
airhead@my.house.net
###
Bone Head
Phone:
(123)123-1234
E-Mail:
###
```

UCF "Practice" Local Contest — Aug 24, 2013

Gas Price Is Going Up/Down

filename: `gasprice`

The gas stations have billboards showing the gas prices (we'll assume only three categories of gas: Regular, Plus, and Super). These billboards display the prices of gas, but the problem is that sometimes some digits are missing from the prices on the billboards!

The Problem:

Given prices for the three categories of gas with at most one digit missing from a given price, you are to determine the exact value of each price. We will assume that Regular is cheaper than Plus which is cheaper than Super. Also assume that Regular is at least $2.00 and Super is at most $5.00.

The Input:

There will be multiple billboards (test cases) in the input file. The first input line contains a positive integer *n*, indicating the number of billboards to be processed. The billboards will be on the following *n* input lines, each on a separate line. Each billboard contains three prices, the first showing Regular, the second representing Plus, and the last showing Super. The first price starts in column one, each price uses three columns (decimal points are not in the input), and there is exactly one space separating prices. The characters used in a price are only digits 0 through 9 and hyphen to indicate a missing digit (there is at most one hyphen per price). Since gas is at least $2.00, the digits 0 or 1 will not appear as the first character for a price in the input. Similarly, the maximum gas price ($5.00) dictates possible valid values for the first character.

The Output:

At the beginning of each test case, output "`Gas Station #`*`g`*`:`", where *g* is the test case number (starting from 1). For each gas station, print the input values and then the output values (each on a separate line and indented three columns). If there are multiple possible (valid) answers, use the lowest valid value for Regular. Then, with the lowest valid value for Regular, use the lowest valid value for Plus. Then, with the lowest valid value for Plus, use the lowest valid value for Super. Assume that input values will always result into at least one possible valid answer.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output. Be sure to line up the output with spaces exactly as given in the Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
2
2-9 285 -99
-50 -99 -99
```

Sample Output:

```
Gas Station #1:
   Input:  2-9 285 -99
   Output: 209 285 299

Gas Station #2:
   Input:  -50 -99 -99
   Output: 250 299 399
```

UCF "Practice" Local Contest — Aug 24, 2013

Dr. Sukthankar's Robot

filename: `robot`

You are taking Dr. Sukthankar's Introduction to Robotics course and are working on your first assignment, to program your new robot to bring Dr. S a cup of coffee. Unfortunately, a fellow student, who decided that he wants your coffee instead, has mis-programmed your robot. Luckily, your robot sends messages to your computer for every move it makes. Using this information, you can deduce where your robot is and direct it to the coffee station and then to Dr. S.

The Problem:

You will be given a listing of the robot's movements on the Cartesian plane. Initially, the robot is located at position (0,0), facing in the direction of the positive x-axis. You will also be given coordinates of both the coffee machine and Dr. S. Your goal will be to first follow the moves the robot has already made, and then give the robot instructions to travel from its current location to the coffee machine and then to Dr. S. The robot understands two types of instructions:

1) Turning (LEFT, RIGHT, UTURN)
2) Moving a specified length in the current direction the robot is facing.

There are further restrictions with respect to what the robot can do:

1) After making the moves following the instructions given in the initial listing (by your mischievous classmate), when moving from the robot's location to the coffee maker, the robot MUST first move in the direction of the x-axis (if necessary), THEN move in the direction of the y-axis (if necessary), to reach the coffee maker. This also applies when moving from the coffee maker to Dr. S.

2) When moving from one location to another, the robot MUST minimize the distance it travels AND the number of commands it takes, given that restriction #1 is satisfied.

The syntax for the instructions given to the robot is as follows:

1) Each command should be on a line by itself.
2) Commands for turning are LEFT, RIGHT and UTURN.
3) The command for moving is MOVE *m*, where *m* represents the number of units to move (in the direction the robot is facing). Note: *m* must be a positive integer.

Note: The left and right turns are always 90 degree turns and the u-turn is 180 degrees. Also, assume that nothing blocks the robot, e.g., the robot can go through the location where Dr. S is to get to the coffee machine (i.e., the robot does not have to go around a block).

The Input:

There will be multiple test cases. The first line of input file contains a positive integer, *n*, representing the number of test cases in the input. The test cases follow. The first line of each test case contains a single positive integer, *k*, representing the number of commands the robot has already executed from its starting position (the origin, facing in the direction of the positive x-axis). The next *k* lines contain the commands the robot has executed. These commands start in column one and there is exactly one space separating the different parts on a line. The list of commands is followed by two ordered pairs on a line separated by spaces, representing the x-y coordinates of the coffee machine and Dr. S, respectively. These four numbers are integers. Assume that the locations of the coffee machine and Dr. S are distinct.

The Output:

At the beginning of each test case, output "`Robot Program #`*p:*", where *p* is the test case number (starting from 1). For each test case, follow all the commands the robot has already processed and print its location. Then, print (give) the commands necessary to take the robot to the coffee machine and then print (give) the commands necessary to take the robot from the coffee machine to Dr. S, complying to the restrictions in the problem statement. Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
3
3
MOVE 100
LEFT
MOVE 50
100 175 200 200
4
UTURN
MOVE 50
LEFT
MOVE 100
-25 50 0 0
1
MOVE 1
2 0 3 0
```

Sample Output:

```
Robot Program #1:
The robot is at (100,50)
MOVE 125
RIGHT
MOVE 100
LEFT
MOVE 25

Robot Program #2:
The robot is at (-50,-100)
LEFT
MOVE 25
LEFT
MOVE 150
RIGHT
MOVE 25
RIGHT
MOVE 50

Robot Program #3:
The robot is at (1,0)
MOVE 1
MOVE 1
```

Notes:
For Sample Input test case 1, a sequence of moves with fewer commands (instructions) will be: MOVE 125, MOVE 25, RIGHT, MOVE 100. But this sequence does not comply to restriction (1), thus is not the desired output.

UCF "Practice" Local Contest — Aug 24, 2013

Brett Favre Diaries

filename: `favre`

The sporting world recently underwent a media overload with the Brett Favre story. The record-breaking Green Bay Packers quarterback, after initially deciding to retire in March, had second thoughts about his decision in the summer. Had Green Bay simply let Brett join the team again, this would not have been much of a story. But, the real saga was quite a bit more complicated. As everyone knows, the media tends to pick individual stories and follows them to no end, much like the O.J. Simpson trial coverage. ESPN treated the Brett Favre story with the same diligence.

You are a football fan and are interested in the Favre story, but enough is enough. You are sick and tired of getting insignificant updates such as, "Brett Favre eats hot dog" on your cell phone. You want to write a program for your cell phone so that it filters Brett Favre news and only reports to you important information, namely whenever he travels to a new city.

The Problem:

Given a list of cities, and a log of chronological information (with a date and time) about Brett Favre, isolate the log entries that contain the name of a city (from the list of cities) different from the city mentioned in the previous log entry with a city. You will be guaranteed that the very first item in the log will contain the name of a city. From that point on, a new log entry is only printed if it contains the name of a city DIFFERENT than the previous log entry with a city (note that the "previous log entry with a city" may not be the "immediate previous log entry"). Your job will simply be to print out the lines of the log that should be presented on the cell phone text feed.

Note: A city is only considered to be in a log entry if it appears EXACTLY, character by character, in the log entry. Namely, capitalization has to match as well as all non-letter characters. However, for our comparison purposes, if the city name is (say) AB and the log entry contains (say) CABD, then the log contains the city, i.e., there doesn't have to be space or punctuation mark before/after the city name in the log.

The Input:

There will be multiple news scenarios (test cases). The first line of the input contains a positive integer n, representing the number of news scenarios to filter. Each scenario follows. The first line of each test case contains a positive integer, m ($1 \le m \le 50$), which represents the number of cities for the test case. The following m lines contain the names of the cities, one per line. These names may contain letters, digits, commas, and spaces (assume no leading or trailing spaces). These input lines (city names) will each be at most 30 characters and will contain at least one non-blank character. Assume that the words Brett and Favre (in any form of upper/lower case) will not be in any city name. Also assume that no city name will be a substring of another city name within a test case. The next input line after the city names will contain another positive integer, k ($1 \le k \le 100$), representing the number of log entries for this case. The next k lines contain one log entry each, given in chronological order (i.e., you don't need to sort them based

on date/time). The log entries can contain any printable characters but will not exceed 100 characters each. Assume that a log entry contains at most one city name.

The Output:

At the beginning of each test case, output "`Brett Log #`*p*`:`", on a line by itself, where *p* is the test case number (starting from 1). The following lines should simply contain, in order, the lines from the entry log that are significant, according to the given criteria.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
2
4
Green Bay
Hattiesburg
New York
Cleveland, Ohio
7
7/23 5:00 PM: Brett leaves Hattiesburg in private plane.
7/23 5:10 PM: Brett eats breakfast omelet on plane.
7/23 5:20 PM: Did I mention Brett is leaving Hattiesburg?
7/23 5:30 PM: Brett falls asleep on plane.
7/23 7:45 PM: Brett lands in Green Bay to waiting fans.
7/23 8:15 PM: Brett watches Packers practice.
7/23 8:20 PM: Deanna is watching practice as well.
2
Cleveland, Ohio
New York
5
7/25 10:00 AM: Brett arrives in New York.
7/25 11:00 AM: Brett does press conference in New York.
7/25 1:25 PM: Brett flies to Cleveland, Ohio.
7/25 4:45 PM: Brett does press conference in Cleveland, Ohio.
7/25 6:00 PM: Brett walks onto field in Cleveland, Ohio.
```

Sample Output:

```
Brett Log #1:
   7/23 5:00 PM: Brett leaves Hattiesburg in private plane.
   7/23 7:45 PM: Brett lands in Green Bay to waiting fans.

Brett Log #2:
   7/25 10:00 AM: Brett arrives in New York.
   7/25 1:25 PM: Brett flies to Cleveland, Ohio.
```

UCF "Practice" Local Contest — Aug 24, 2013

In the Spotlight

filename: `spot`

Starlet Stacie always insists that the spotlights must shine upon her sufficiently, regardless of where she stands on the stage. Otherwise, she makes a scene.

Consider the floor of the stage to be a Cartesian plane, the front of the stage is the *x*-axis and the sides of the stage are at $x = 0$ and $x = x_{max}$, with the *y*-axis going toward the back of the stage.

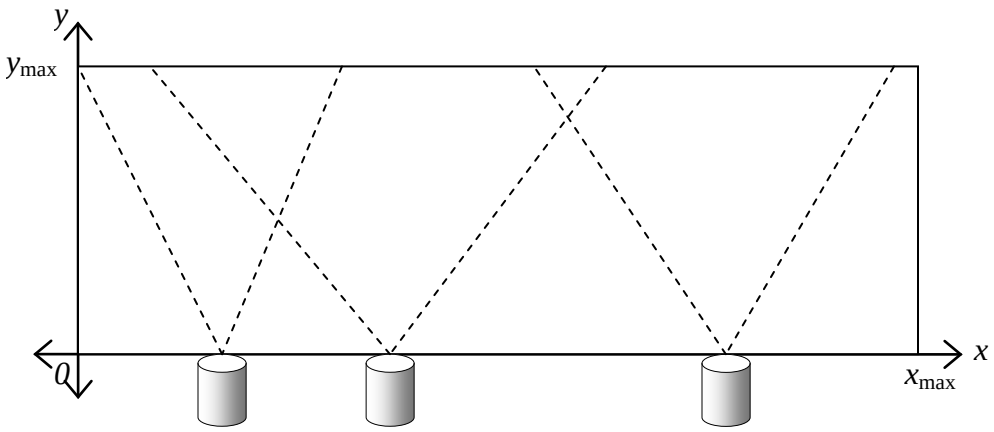


Fig. 1: stage floor and spotlights, with superimposed Cartesian axes

So Stacie's stage position is always $0 < x < x_{max}$ and $0 < y < y_{max}$. The spotlights are all mounted along the *x*-axis, and are all aimed onto the stage (parallel to the *y*-axis), though they might have different focus angles. Height above the stage floor isn't important, so we need to consider only the 2-dimensional plane.

The focus angle of each spotlight is an angle relative to the aimed direction, on either side, and indicates the area covered by that light beam. Thus each spotlight will essentially cover a stage area that is an isosceles triangle (with the back of the stage as the 3rd side).

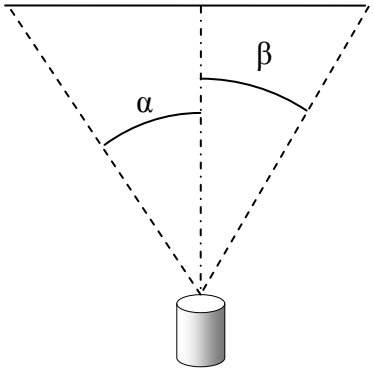


Fig. 2: focus angle = α = β

Anything outside a spotlight's triangular coverage area gets negligible illumination. The intensity of a light within the triangle (any point within .01 degrees of the focus angle is considered inside the triangle) is given by $I_{source} \div d^2$ where I_{source} is the intensity of the light at its source point, and d is the distance from the source point.

If Stacie is standing within the coverage area of more than one light, the total intensity of light shining on her is simply the sum of all such lights.

The Problem:

Given Stacie's stage position and the positions and intensities of various spotlights during a scene, find the intensity of light shining on Stacie.

The Input:

The first line of input will contain only a positive integer *p*, which is the number of scenes to evaluate. This will be followed by *p* scene descriptions. The first line of each scene description will contain three integers, x_S and y_S ($0 < x_S, y_S < 1000$), representing Stacie's position, and *n* ($0 < n < 100$), the number of lights that are turned on. The second line will contain *n* distinct non-negative integers less than 1000: $x_1, x_2, \ldots x_n$; these are the positions of the lights along the *x*-axis. (Note that distance units are not provided because all distance units are the same.) The third line will contain *n* positive integers less than 90: $\alpha_1, \alpha_2, \ldots \alpha_n$; these are the focus angles, in degrees, for each corresponding light on the preceding line. The fourth line of each scene description contains *n* positive integers less than 10,000: $I_1, I_2, \ldots I_n$; these are the intensities of the lights at their source points for each corresponding light on the preceding lines. All numbers on the same line will be separated from each other by exactly one space, with no leading or trailing spaces. (Use 3.14159265 for the value of π.)

The Output:

For each scene description, output a message of the form

`Scene #s: Spotlight intensity on Stacie is t`

where `s` is the number of the scene in the input (counting from 1) and `t` is the total light intensity, rounded to the nearest three decimal places (examples: 1.2374 would round to 1.237 and 1.2375 would round to 1.238).

Leave a blank line after the output for each data set. Follow the format illustrated in Sample Output.

Sample Input:

```
2
5 5 3
0 5 10
50 45 40
100 100 100
10 2 2
10 40
15 5
400 500
```

Sample Output:

```
Scene #1: Spotlight intensity on Stacie is 6.000

Scene #2: Spotlight intensity on Stacie is 100.000
```

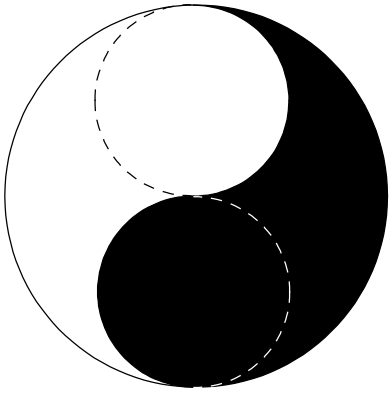
UCF "Practice" Local Contest — Aug 24, 2013

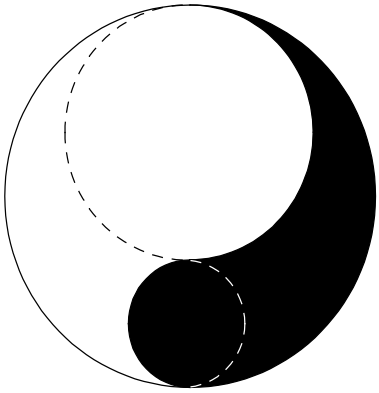
Ty G. Too?

filename: `yinyang`

The Chinese Taijitu symbol, also known as the Yin and Yang, symbolizes the duality and balance of everything. It appears in many contexts, including ancient Confucian writings, martial arts, and even the national flag of South Korea. The Taijitu is always shown with the yin (dark side) and the yang (light side) in balance. But, what happens when they are unbalanced?

The typical Taijitu has the dark (yin) and light (yang) side separated by a specific curve that is related to the apparent path of the sun around the earth (the ecliptic), but for the purposes of this problem, we'll simply use two connected semicircles as shown below. In the image on the left, the Taijitu is balanced, with an equal area of yin and yang. The image on the right has significantly more yang (because the upper semicircle has a larger radius than the lower semicircle), and is therefore unbalanced.





Note: Each side is usually depicted with a "seed" (a small dot) of the opposite color. For simplicity, we will ignore this.

The Problem:

Given the overall radius (in centimeters) of the Taijitu, and the radius (in centimeters) of the upper semicircle (as shown in the images above), determine the total area of yin and yang in the corresponding Taijitu. Note that the yang (the light side) always appears on the left side of the Taijitu, and the two semicircles defining the division are always oriented as shown above.

Use a value of 3.14159 for π (pi).

The Input:

There will be multiple Taijitu(s). Input begins with an integer, n ($1 \le n \le 50$), on a line by itself, indicating the number of Taijitu(s) to examine. On each of the next n input lines will be two integers, a and b ($1 \le b < a \le 100$), where a represents the overall radius of the Taijitu, and b represents the radius of the upper semicircle.

The Output:

For each Taijitu in the input, print a line of the form "`Taijitu #`*t*`: yin `*x*`, yang `*y*", where *t* is the Taijitu number (starting from 1), *x* is the area of the yin (the dark side) in square centimeters, and *y* is the area of the yang (the light side) in square centimeters. Round the area to the nearest two decimal places (examples: 1.274 would round to 1.27 and 1.275 would round to 1.28). Leave a blank line after the output for each data set. Follow the format illustrated in Sample Output.

Sample Input:

```
2
10 5
12 7
```

Sample Output:

```
Taijitu #1: yin 157.08, yang 157.08

Taijitu #2: yin 188.50, yang 263.89
```