First Annual
University of Central Florida

High School Programming Tournament: Online Edition

Problems

Problem Name	**Filename**
Rail Fence Cipher	cipher
Driving in Circles	circles
Epic Winn-Dixie	epic
Say Cheese!	framing
Logic Puzzles	logic
Magic Numbers	magic
No Odd Rooms!	odds
Unbelievably Covered, Fast Parking	parking
Spaceman Spiff	spiff

Call your program file: *filename*.c, *filename*.cpp, or *filename*.java

For example, if you are solving Magic Numbers:
Call your program file: magic.c, magic.cpp or magic.java
Call your Java class: magic

Rail Fence Cipher

Filename: `cipher`

The Rail Fence Cipher is a way of encrypting a string. It works by arranging letters on a zigzag pattern on *n* rows of letters, and then reading the letters of each row. For example, the string "RAIL FENCE CIPHER" can be encoded by removing all characters except letters and digits, then arranging the letters along three rows like this:

```
R . . . F . . . E . . . H . .
. A . L . E . C . C . P . E .
. . I . . . N . . . I . . . R
```

We read the letters off by the rows, in groups of 5, so the encrypted version (ciphertext) is "RFEHA LECCP EINIR". To increase the security even more, we can start reading the rows from a particular number, *r*. For example, the same string "RAIL FENCE CIPHER" with *n*=4 and *r*=3 looks like this:

```
3: R . . . . . N . . . . . H . .
4: . A . . . E . C . . . P . E .
1: . . I . F . . . E . I . . . R
2: . . . L . . . . . C . . . . .
```

When read off the ciphertext is "IFEIR LCRNH AECPE".

The Problem:

Given a message, output the message encrypted after running the Rail Fence Cipher.

The Input:

The first line of each message consists of two positive integers, *n* and *r* ($1 \leq n \leq 100$; $1 \leq r \leq n$). The next line will be a message to encrypt. The message will not exceed 100 characters in length and will contain only letters, numbers and symbols. The last message will be followed by the line "0 0" and should not be processed.

The Output:

For each message, you must remove all non-letters and non-digits from the input string. Then you must encrypt it using the Rail Fence Cipher with *n* rows. Finally, output the cipher text, offset by *o* rows, grouping the letters into sets of 5 with a space between groups (the last group in the message may have fewer than 5 characters) and printing a period after the ciphertext.

Sample Input:

```
3 1
RAIL FENCE CIPHER
4 3
rail fence cipher
4 3
rail fence ciph
0 0
```

Sample Output:

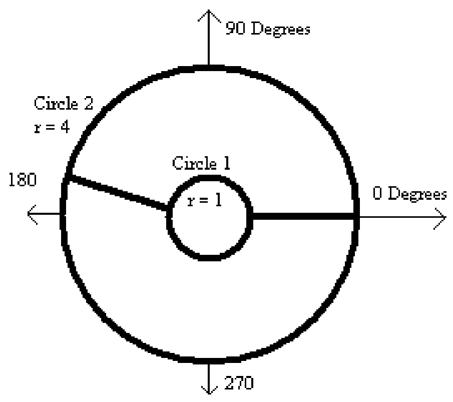
```
RFEHA LECCP EINIR.
ifeir lcrnh aecpe.
ifeil crnha ecp.
```

Driving in Circles

Filename: `circles`

The University of Central Florida (UCF) has a unique layout that makes it more interesting to drive through. Specifically, the roads form circles around the Student Union, with radial cross streets connecting circles. The whole setup can be disorienting for the delivery trucks that need to bring in packages to the center of UCF, since most drivers are not used to driving in circles when they already know where they are going. Luckily, no street or road on campus is one-way so drivers avoid that confusion.

As a new driver for a delivery company that includes UCF and other similarly arranged locations, Ali is finding that it can be difficult to figure out the fastest way to get to a delivery with all the circles. Determined to minimize the distance he has to travel to get to a delivery point, Ali decides that he'll make an application for his new smart phone to help him out. You just put in the layout of the roads and streets and it tells you how far you have to travel. Who knows, maybe there will be a market for just such an app!



The Problem:

Given a description of the map as well as starting and ending locations, determine the shortest trip.

If you need a value for π, use the one provided by your environment or 3.141592653589793 if one is not provided.

The Input:

The first line of the input will contain a positive integer, *t* ($t \leq 100$), giving the number of trips to follow. Each trip begins with a line containing the integers, *c* ($1 \leq c \leq 10$) and *r* (c-1 $\leq r \leq 20$), where *c* is the number of circular roads that surround a single center point and *r* is the number of radial streets connecting the circular roads. The next line will contain *c* integer circle radii in strictly increasing order, each separated by a single space.

The next *r* lines will each contain two integers, separated by a single space, describing a radial street, *d* ($1 \leq d < c$) and *a* ($0 \leq a < 360$). *d* is the one-based index of the starting circular road and represents connecting circular road *d* with next larger circular road *d*+1. *a* is an integer angle in degrees in relation to the center of all circular roads, with 0 degrees pointing directly to the East, and the angle increasing in the counter-clockwise direction (90 degrees would be North, etc.).

All circular roads will be connected by radial streets, and it will always be possible to reach one circular road from another. Radial streets will only connect to valid circular roads. No road or street will lie on top of another road.

After the *r* radial street descriptions, there will be four more integers on a single line, *sc sa fc fa* ($1 \leq sc$, $fc \leq c$ and $0 \leq sa$, $fa < 360$), denoting the starting and ending locations. *sc* is the one-based index of the starting circular road, and *sa* is the angle of the starting point compared to the center of the circular road. *fc* and *fa* are the destination circular road and the angle of the point on the destination circular road, respectively. Again, angles are in degrees and given in the same orientation as the radial street descriptions (0 degrees is East, 90 is North, etc.).

The Output:

For each trip, print a line containing only the shortest distance from the starting point to the ending point, traveling along circular roads and radial cross streets. This distance should be output rounded to two decimal points (for example, 2.435 should be rounded to 2.44 and 2.934 should be rounded to 2.93).

Sample Input:

```
2
1 0
1
1 0 1 180
2 2
1 4
1 0
1 170
2 180 1 90
```

Sample Output:

```
3.14
5.09
```

Epic Winn-Dixie

Filename: `epic`

Winn-Dixie Marketplace has noticed the recent rise in super stores such as Super Target and Super Walmart and decided to get in on the action. The chain has decided to appeal to a younger crowd by calling their store Epic Winn-Dixie. To continue with the theming of the new store they have also decided to change all items in the store with precisely “Win” and “Fail” at the start of their names to have the prefix “Epic”added to them.

The Problem:

Given all the items in the store, read in each grocery item and output the list of items after the theme change to Epic Winn-Dixie.

The Input:

First, you will be given the number of items that must be changed, *n*, on a line by itself. That line will be followed by *n* lines containing grocery items that are in the the store. Each item will be a string of characters whose length is at least 1 and at maximum 80. Each line will not have any leading or trailing spaces.

The Output:

For each grocery item print out a line containing a potentially modified grocery item that either has no prefix or the prefix “`Epic `” in front of it if required by the rules presented.

Sample Input:

```
7
Windex
Fail Sauce
Winston Churchill's Famous D-Day Donuts
Monster
1337 Sauce
Buffalo Wings
Fail Buckets
```

Sample Output:

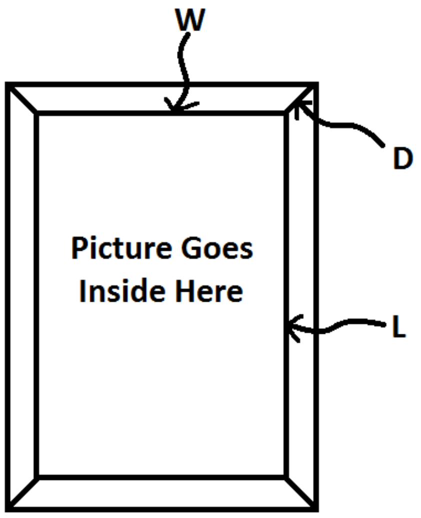
```
Epic Windex
Epic Fail Sauce
Epic Winston Churchill's Famous D-Day Donuts
Monster
1337 Sauce
Buffalo Wings
Epic Fail Buckets
```

Say Cheese!

Filename: `framing`

The Ukrainian Caroling Federation loves taking photos. They like small pictures, large pictures, and everything in between. In fact, they've just gotten back from the annual Christmas retreat, and they have thousands of photos from everyone in the group. Being a new member of the group, people don't respect you very much. But you aim to change that this year! You've had a great idea for Christmas presents to make everyone love you. Because everybody loves their pictures so much, they of course would love to hang these pictures around their house. Naturally, everyone is going to need frames, and lots of them.

To make the gifts nicer, you decide you're going to make all of the frames yourself. You already have a large amount of wood and have decided that every frame you make will be of the same general shape, as seen in the picture. *L* and *W* are the length and width of the picture you are framing, and *D* is the length of the diagonal from the corner of the picture to the corner of the frame (all measurements given in inches). So that the frame fits together properly, each of these should meet at a 45° angle. For simplicity, you may assume that you will build the frame to go perfectly around the picture and that this is the only wood you will need (you will use other materials to fix the picture inside the frame). Furthermore, each frame you build will be 1 inch deep.



Now you want to know: how much wood will you need to make all the frames?

The Problem:

Given *L*, *W*, and *D*, determine how much wood is needed to make the frames for everyone in your group. Because you may do this sort of thing again in the future, your program should be able to calculate this for multiple groups.

The Input:

There will be multiple groups in the input file. The first input line contains a positive integer, *t*, indicating the number of groups to be processed. This will be followed by *t* groups. Each group will begin with an integer, *n* ($1 \le n \le 100$), indicating the number of people in the group. This is followed by *n* group members. Each group member starts with an integer, *f* ($1 \le f \le 100$) on a line by itself, indicating the number of frames to make for that group member. Following this will be *f* lines, each containing 3 integers *l*, *w*, and *d* ($1 \le l,w,d \le 100$). Each integer will be separated by a single space, and no line will contain leading or trailing spaces.

The Output:

At the beginning of each group, output "`Group #x: v cubic inches`" where x is the group number (starting from 1), and *v* is the total volume of the wood needed to build all of the frames for everyone in the group rounded to 2 decimal places (2.485 should be rounded to 2.49, 2.484 should be rounded to 2.48). Leave a blank line after the output for each test case.

Sample Input:

```
2
1
1
4 6 1
2
2
4 6 2
5 7 2
1
8 10 1
```

Sample Output:

```
Group #1: 16.14 cubic inches

Group #2: 105.68 cubic inches
```

Logic Puzzles

Filename: `logic`

Do you like logic puzzles?! What is a logic puzzle, you ask? Well, let me tell you.

In each puzzle you are given a series of categories, and an equal number of options within each category. Each option is used once and only once. Your goal is to figure out which options are linked together based on a series of given clues. Each puzzle has only one unique solution, and each can be solved using simple logical processes (that is, educated guesses are not required).

For example, let's say you have the following logic puzzle where you have three names, three homes and three ages, and you must connect the name with the home and the age given clues:

Age	**Name**	**Home**
14	Alice	Orlando
21	Bob	Paris
68	Eve	Venice

Clues

1. Bob is 21 years old.
2. The one who lives in Paris is older than Eve.
3. Eve is younger than Bob.
4. Alice does not live in Orlando.
5. The three people are Alice, the 21 year old and the one who lives in Venice.

Given the clues, using only logical deductions you can solve the puzzle. The solution is:

> Eve is age 14 and lives in Venice. Bob is age 21 and lives in Orlando. Alice is age 68 and lives in Paris.

In this problem, every logic puzzle will include an age category. The other categories (there may be more than 2 of them) can be anything. Additionally, there may be more than 3 options for each category. Finally, you will always be given one clue in the form of Clue #5 from the example. That is, a clue in which each person listed is different from everyone else listed.

After solving a number of logic puzzles by hand, you decide to impress your friends by writing a program to solve them for you. Well, what are you waiting for? Go impress some people!

The Problem:

Given the description of a logic puzzle, determine the unique solution by following all of the specific rules. Since new logic puzzles come out every day, your program should be able to handle multiple puzzles.

The Input:

There will be multiple logic puzzles in the input file. The first input line contains a positive integer, *t*, indicating the number of puzzles to be processed. This will be followed by *t* puzzles. Each puzzle will start with a line containing two numbers, c ($1 \le c \le 4$) and o ($1 \le o \le 5$), giving the number of categories and the number options in each category, respectively. Following this will be a line containing *c* categories. The first category listed will *always* be "Age." After this will be *c* lines with *o* options on each line. The first category will always list *o* increasing positive integers. Each option will be used exactly once. Next will be a line containing a positive integer k ($1 \le k \le 100$), giving the number of clues. This will be followed by *k* clues of the form:

o_1 s o_2

where *s* can be as follows:

<	person with o_1 is younger than person with o_2
>	person with o_1 is older than person with o_2
=	person with o_1 is the same as person with o_2
!	person with o_1 is not the same as person with o_2

Finally, there will be a line containing exactly *o* options. This is the list of *o* unique properties from the solution. No line will contain leading or trailing spaces, and all space-separated values will be separated by exactly one space. Each category name or option will contain at least 1 and no more than 20 case-sensitive letters (except for Age which is a positive integer).

No line in the input will contain more than 80 characters. Note that the first puzzle in the Sample Input maps to the example from the problem statement.

The Output:

At the beginning of each puzzle, output a single line "`Logic Puzzle #x:`" where *x* is the puzzle number (starting from 1). Then, on a separate line echo the categories for the puzzle exactly as they appear in the input. This should be followed by *c* lines, each containing *o* options, giving the solution. The lines should be listed in order of increasing age. The options on each line should be listed in order of the categories given and separated by a single space. Leave a blank line after the output for each puzzle.

Sample Input:

```
2
3 3
Age Name Home
14 21 68
Alice Bob Eve
Orlando Paris Venice
4
Bob = 21
Paris > Eve
Eve < Bob
Alice ! Orlando
Venice Alice 21
2 2
Age Name
17 34 49
David Timothy
1
David > Timothy
Timothy 34
```

Sample Output:

```
Logic Puzzle #1:
Age Name Home
14 Eve Venice
21 Bob Orlando
68 Alice Paris

Logic Puzzle #2:
Age Name
34 Timothy
49 David
```

Magic Numbers

Filename: `magic`

Choose a number between 1 and 10, inclusive. Now multiply that number by 2. Now add 6. Now divide that number by 2. Now subtract your original number. The number you are thinking of now is 3.

This "magic trick" is just a simple math formula and you can substitute anything for 2 and anything for 6 and the trick will still work, you will just have a different number instead of 3. However, in order for the trick to really work, the arithmetic involved must be simple (since the person is doing it in their head). In particular, no intermediate (or final) result may exceed 144 or be negative, and all intermediate (and final) results must be integers.

The Problem:

Given a description of the magic trick, you must output whether the trick is a good one, or if it is too difficult to do.

The Input:

Each magic trick consists of two non-negative integers, *x* and *y*. The magic trick is "Choose a number between 1 and 10, inclusive. Now multiply that by *x*. Now add *y*. Now divide that number by *x*. Now subtract your original number. The number you are now thinking of is *z*." The last trick will be followed by a line with two zeroes, which should not be processed.

The Output:

For each magic trick, you must output "The number left over is *n*." where *n* is the number that is left after doing the trick. However, if the trick is too difficult to do, then output "The trick is too difficult to do."

Sample Input:

```
2 6
3 4
0 0
```

Sample Output:

```
The number left over is 3.
The trick is too difficult to do.
```

No Odd Rooms!

Filename: `odds`

Jimmie Flowers has a bit of an obsessive-compulsive order. He just can't stand odd numbers! Unfortunately, Headquarters is moving to a new building and offices are being re-assigned. He needs to help his boss determine a good office for him (or else, his productivity will be greatly affected!).

The Problem:

Given a number, determine whether it is odd or even.

The Input:

The first line of the input will contain a single positive integer, *n*, representing the number of offices that are still available (some of Jimmie's superiors have already been assigned their offices). Following this, on each of the next *n* lines there will be a single, positive integer by itself, representing the number of an available office. No office number will appear more than once.

The Output:

For each available office, determine if it has an odd or even number. If it is even, then output the number on a line by itself. If it is odd, then do not output anything (the output will be a list of offices with even numbers – which will make Jimmie quite happy!).

Sample Input:

```
8
142
267
116
373
165
183
282
300
```

Sample Output:

```
142
116
282
300
```

Unbelievably Covered, Fast Parking

Filename: `parking`

Jack Johnson and John Jackson love racing to class. Lately, John has consistently beat Jack to class because he leaves the parking garage before Jack. Each floor of the garage is a rectangle and each floor is perfectly aligned with the first floor. The parking garage has exits in each corner of the first floor of the garage. Stairs that connect all floors are located in the four corners of each floor. The number of seconds it takes Jack to reach the exit or stairs (on the same level) from his parking spot is the *manhattan distance* from that spot to the exit or stairs. The manhattan distance is defined as the sum of the absolute values of the difference in coordinates. For example, consider a spot at (a,b) and an exit at (x,y), the manhattan distance from the spot to the exit is | a - x | + | b - y |. It takes Jack two seconds to go down one floor. In order to beat John, Jack has asked for your help. Of course, Jack gets a different parking spot each day so he needs a solution that can work for multiple days.

The Problem:

Given the integer dimensions of a garage, the number of floors the garage has, and the location of available parking spots in the garage, determine the location of the parking spot that will get Jack to an exit of the garage in the minimal time.

The Input:

The first line of the input will begin with a single positive integer, *n*, that represents the number of days to be handled. Each day will begin with four integers, *l w f k* ($1 \le l, w, f, k \le 1000$), representing the length of the garage, width of the garage, number of floors in the garage, and the number of available spots, respectively. Each of the next *k* lines will contain three integers, *a b c* ($0 \le a \le l$, $0 \le b \le w$, $1 \le c \le f$), representing the coordinate of the spot in relation to the length, the coordinate of the spot in relation to the width and the floor the spot is on, respectively. The corners of the garage are (0, 0), (*w*, 0), (*w*, *l*) and (0, *l*). You are guaranteed that no two spots will be in the same location and no spot will be located at the exit or any stairs.

The Output:

For each day, output one line in the following format: "`Day #k: x y z t`" where *k* is the number of the day starting at 1, *x y z* is the location of the closest spot to an exit (representing the coordinate of the spot in relation to the length, the coordinate of the spot in relation to the width and the floor the spot is on), and *t* is the time in seconds it takes to get to the exit of the garage. Within each day, you will be guaranteed that only one parking spot will be the best spot for getting Jack to an exit the quickest.

Sample Input:

```
2
4 4 4 3
2 2 1
2 3 1
0 1 4
1000 1000 500 2
500 500 2
1 1 200
```

Sample Output:

```
Day #1: 2 3 1 3
Day #2: 1 1 200 400
```

Spaceman Spiff

Filename: `spiff`

Spaceman Spiff is off exploring the outer regions of Mars when his shuttle unexpectedly malfunctions! Luckily his transponder still appears to be working so Spiff sends a quick message to mission control asking for help. The response he gets back from mission control is shown below:

sresp eht der button

Oddly enough, the response also appears to be jumbled and he needs the information to correct his ship's course before his imminent crash. All of the characters appear to be correct but the ordering of characters was messed up in transmission. Being the quick thinker that he is, Spiff picks one of the words ("eht") and quickly pulls out his trusty dictionary, looks up all three-letter words containing the characters e, h, and t and locates the word "the", so he rearranges "eht" to read "the". He then realizes that "der" can be rearranged in this same pattern (i.e., first letter becomes the third letter, the second letter stays the same, and the third letter becomes the first) to make the word "red". All of the three-letter words are following the same pattern! However, he notices that he must find a different pattern for each of the other lengths. Following this technique for the rest of the words in his dictionary, Spiff soon realizes that the intended messages was:

press the red button

Being the intuitive AI that you are, you decide to correct these mistakes for Spiff for all upcoming transmissions by requesting that Spiff input his dictionary so that you can translate the message from mission control before displaying it to Spiff.

All incoming words of length *k* will use the same pattern to unscramble them, so your job is to determine the correct pattern for every word of length *k* ($1 \le k \le 7$) by looking for matching words using the exact characters as the scrambled word in Spiff's dictionary. Note that you can only assume an unscramble is correct if the letters only translate into exactly one word in the dictionary. That is, if the message is, "won tih eht der eno" and the dictionary contains the words "now", "won", "one", "hit", "red" and "the", then you cannot determine whether the first word should be "won" or "now" and must look for the pattern for 3 letter words elsewhere. In this case, "eht" can only be translated into "the" and using this pattern the rest of the 3 letter words, the message will be translated into, "now hit the red one"

The Problem:

Given a dictionary and scrambled messages of words, unscramble the messages.

The Input:

There will be multiple malfunctions to handle. Each one will begin with a line containing a single integer, n ($1 \leq n \leq 50$), representing the number of words in the dictionary. The following n lines will each consist of a single word of no more than 7 letters. Following the dictionary is a line with a single integer, c ($1 \leq c \leq 50$), representing the number of words in the scrambled message. On the following c lines is one word of the message (again, no more than 7 letters). Input ends with a line containing a zero, which should not be processed.

All words will be contain characters a-z and 0-9, with no capital letters or punctuation.

For every malfunction, at least one word of each length will map to one and only one distinct dictionary word. Furthermore, at least one of these words mapping to one and only one distinct dictionary word will not have any ambiguous unscramble, where an ambiguous scramble occurs when a word contains two of the same letter. For example, "aba" unscrambled into "baa" is ambiguous because which character "a" goes to which location is not known with certainty.

The Output:

For each message, first output a header "`Message #n: `" where n is the number of the message (starting with 1). Then, on the same line output the unscrambled message for Spaceman Spiff. Follow the output for each message with by a blank line.

Sample Input:

```
2
hi
world
2
ih
wordl
3
now
won
hit
5
won
tih
eht
der
eno
0
```

Sample Output:

```
Message #1: hi world

Message #2: now hit the red one
```