Twelfth Annual
University of Central Florida

ACM · UPE

High School Programming Tournament

Problems

Problem Name	Filename
Banana Man	BANANA
Bit-Block Characters	BITS
As the Sign Turns	SIGN
Elelator Go Down the Hole!	ELEVATOR
	CHANGE
Spice Up Your Life!	PEPPER
Benjamin's Bubbles	BUBBLES
Jump Puzzle	JUMP
Jason and His Friends	FRIENDS
Dictionary Sorting	SORT

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

For example, if you are solving Spice Up Your Life!:
Call your program file: PEPPER.PAS, PEPPER.C or PEPPER.CPP
Call your input file: PEPPER.IN

Banana Man

Filename: BANANA

Ali grows bananas at an oasis in the desert. Harvest time has come and Ali has had a successful crop of 100,000 bananas. Now he needs to get them to market 500 miles away, but he has a bit of a problem he would like your help with. To get his bananas to market he must use his camel. This camel can carry a maximum of 500 bananas, but must be coerced with 1 banana for every mile it travels. To get his product to market with any left, Ali will have to move the bananas in several steps. Ali would like a program which will tell him how many bananas he gets to market given how far he travels in-between stops.

For example, suppose Ali goes 100 miles in-between stops. On his first trip, he will load up his camel with 500 bananas. He must feed the camel 100 bananas to get to the first midway point. There, he will drop off 300 bananas, leaving 100 for the trip back. After moving bananas to this first stop, he will have 60,100 bananas left. By the time he gets to market, he will have 7,900 bananas left.

By trying out different values Ali can determine what the best approach will be.

The Input:

There will be multiple input sets. Each input set consists of one positive integer on a line by itself for the distance in-between stops. Input will be terminated by end of file.

The Output:

For each input case, output the number of bananas Ali gets to market in the form "By moving the bananas # miles per step, # bananas get to market."

Sample Input:

```
100
96
```

Sample Output:

```
By moving the bananas 100 miles per step, 7900 bananas get to
market.
By moving the bananas 96 miles per step, 8300 bananas get to
market.
```

Bit-Block Characters

Filename: `BITS`

Many computer video subsystems use bit-blocks to describe how their text mode characters appear on the screen. One way to implement this is to use an 8 by 8 bit block, defined by a string of eight 8-bit unsigned integers (0..255). The binary representation of each integer corresponds to a vertical column in the bit-block matrix. The least significant bit of each integer is at the top of the matrix and the most significant bit is at the bottom.

For example, the bit block defined by the string:

248 20 18 17 17 18 20 248

Would look like the upper-case letter "A":

```
...**...
..*..*..
.*....*.
*......*
********
*......*
*......*
*......*
```

The Input:

On the first line of the input file will be a single integer N, $1 \le N \le 8$. On the next N lines will be eight 8-bit unsigned integers describing a bit-block.

The Output:

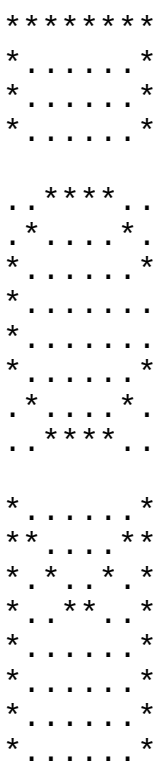
For each string of integers, print the corresponding bit-block character in an 8 by 8 character matrix. Use a period "." where a "0" would be and an asterisk "*" where a "1" would be. Skip one line between each matrix.

Sample Input:

```
3
248 20 18 17 17 18 20 248
60 68 129 129 129 129 68 36
255 2 4 8 8 4 2 255
```

Sample Output:

```
...**...
..*..*..
.*....*.
*......*

.**....*
*..**..*
*..**..*
.**....*

********
*......*
*.....*.
*....*..
*...*...
*..*....
*.*.....
**......
```



As the Sign Turns

Filename: `SIGN`

Mike was sitting in a restaurant one evening when his friends saw a humorous sign and pointed it out to him. He looked but couldn't see it. He asked "where?" and all his friends replied "right there!" However, Mike still couldn't see it. Mike didn't see it because this sign both turned (rotated) and blinked on and off. Whenever it was facing him, it was off and whenever it was facing away from him, it was on!

The Problem:

Given the rotating and blinking rates of the sign, determine whether Mike can "see" the sign at requested times. For this problem, we say Mike can see the sign if his viewpoint has an angle of incidence greater than 10 degrees with the sign at any time it is on. The angle of incidence is defined to be the angle between Mike's viewpoint of the sign and the sign itself (see the picture below).

<picture>

If his angle of incidence is greater than 10 degrees but the sign is off, then he still cannot see it. Assume that Mike sits at the origin and the sign is on the y-axis. Furthermore, the sign always starts perfectly facing towards him but always off (i.e. he can't see it immediately at the start).

The Input:

The first line will contain an integer n representing the number of data sets in the file. For each data set, there will be a line containing the rotating and blinking rates for a sign, respectively. The rotation rate is the amount of time the sign uses for one complete rotation. The blinking rate is both the amount of time the sign is on as well as the amount of time that the sign is off (for example, a blinking rate of 3 means that the sign will be off for 3 seconds, then on for 3 seconds, etc.). Following the rates, there will be a line containing a single integer t. On the next t lines, there will be one floating point number representing a time at which to test if Mike sees the sign.

The Output:

For each sign, print a header "Sign #i:" where i is the ith sign in the input. Following that, tell whether Mike can see the sign at each requested times. If he can, output "Mike sees it!" If he never sees it, output "Mike is blind." Leave a blank line after the output for each sign.

Sample Input:

```
2
<rotation_rate> <blinking_rate>
3
0.0
0.5
```

```
1.3
<rotation_rate> <blinking_rate>
2
15.2
18.1
```

Sample Output:

```
Sign #1:
   Mike is blind.
   Mike sees it!
   Mike is blind.

Sign #2:
   Mike is blind.
   Mike sees it!
```

Elelator Go Down the Hole!

Filename: ELEVATOR

Herman is a building inspector in Turvyville. His job is to make sure that the elevators work like they should. Of course, Turvyville has weird rules regarding its elevators (otherwise this would be an easy problem!). Not every elevator goes to every floor. Actually, most don't. The only requirement is that every floor be reachable by elevators. Now, Herman is a lazy inspector - otherwise he wouldn't need your help - and doesn't want to ride the elevator all day. In fact, he doesn't even check that every floor is reachable. He just checks a couple of random floors to make sure that you can get from one to the other, and grades the building based on that. Your job is to take the plan provided by the architect and check for a path between two specified floors.

In his couple of days of actually doing work, Herman has stumbled onto a simple method of searching for a path. Suppose that he is inspecting the building represented in the Sample Input below. He is looking for a path between the 4th and 10th floors. He starts on floor 4, marks it as visited on his clipboard, and checks his plans. The first floor he sees that he can get to from 4 is floor 1. He takes the elevator down to 1, and marks it as visited on his clipboard. On floor 1, he checks his plans again. The first floor he sees is floor 4, but he notes that he's already been there, so he goes on to floor 9. At each floor, he repeats the process, winding through floors 8, 12, and 7 before ending up on floor 11. Once there, he checks his plans and sees that he can only get to floors 7 and 9, both of which he has already visited. So he backtracks over the path he came in on. From floor 7, he can get to 11 and 12, both of which he's already seen. So he goes back again. From 8, he can only go to 9 and 12, so he backs up again. From 9, he's already been to 1 and 7, but not 10. He takes that elevator, marks the floor as visited, and realizes that he's reached his destination. So much work!

The Input:

There will be multiple sets of data in the input. The first line of each set will be *n*, a positive integer less than 10, which is the number of floors in the building that Herman is inspecting. On the next *n* lines will be *n* integers, each either a 0 or a 1. Each line represents a floor in the building, and each number in the row represents the other floors. A 0 in a floors position indicates that it is impossible to get to that floor *in one elevator ride* from the current floor. A 1 indicates that it is possible. Following this matrix will be a line containing a single positive integer *m*. The next *m* lines will be floor combinations that Herman wants to test - the first number on each line will be the starting floor, and the second number the desired destination. End of data is indicated by a negative number of floors.

The Output:

For every data set, first print the message "Building #x:" with appropriate x. On the following lines, print each of the floor combinations he checked, in order, in the form "a b: s", where a and b are the floor combinations he checked, and s is either the string "Possible" or "Not Possible", as appropriate. There should be a blank line after the output for each building.

Sample Input:

```
12
0 0 0 1 0 0 0 0 1 0 0 0
0 0 0 0 1 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 1 0 0 0 0
0 1 0 0 0 1 0 0 0 0 0 0
0 1 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 1 0 0 0 0 1 0 0 1
1 0 0 0 0 0 0 1 0 1 1 0
0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 1 0 0 0
0 0 0 0 0 0 1 1 0 0 0 0
5
4 10
7 10
6 8
2 6
3 12
-1
```

Sample Output:

```
Building #1:
4 10: Possible
6 8: Not Possible
2 6: Possible
3 12: Possible
7 10: Possible
```

CHANGE

Filename: CHANGE

Spice Up Your Life!

Filename: PEPPER

Joel is violently allergic to pepper. So allergic, in fact, that if he even *hears* the word "pepper" too much, he sneezes. Your problem is to screen sentences that Joel might hear, and determine if they will make him sneeze.

The Input:

The first line of the data file will be a single integer x ($0 \le x < 100$), representing the number of data sets in the file. The first line of each data set will be two integers, n ($0 \le n < 10$) and m ($m \ge 0$), where n is the number of times that Joel can hear "pepper" without sneezing, and m is the number of sentences in the set. The rest of the the set will consist of one sentence per line. Each sentence consists of valid alphanumeric characters and punctuation, and will be no longer than 70 characters.

The Output:

For every data set, first print the header "`Data Set #i:`", with appropriate *i*, followed by a blank line. For every sentence in the set, print the sentence and, on the next line, print either "Joel does not sneeze." or "Ah-choo!", as appropriate. Note that since these are sentences that Joel is to *hear*, case does not matter. Also, "pepper" does not need to be the complete word; he reacts just as violently to "peppering" or "repepper". Each sentence should be separated by a blank line, and each data set should be separated by two.

Sample Input:

```
1
0 4
"PEPPER" is the name of this problem.
Joel is not allergic to salt.
The hamburger I had for lunch was lightly peppered.
The end of the input is near!
```

Sample Output:

```
Data Set #1:

"PEPPER" is the name of this problem.
Ah-choo!

Joel is not allergic to salt.
Joel does not sneeze.

The hamburger I had for lunch was lightly peppered.
Ah-choo!

The end of the input is near!
```

Joel does not sneeze.

Benjamin's Bubbles

Filename: BUBBLES

Benjamin likes to blow bubbles. He recently obtained a new bottle and started blowing bubbles in the house. Unfortunately, his parents have carpeted tiles that are easily ruined by the bubble solution. Needless to say, his parents were not happy. Please help Benjamin and his parents figure out which tiles were ruined so that they can be replaced.

The Problem:

Given a description of a room's carpet tiles and a set of bubbles, determine which carpet tiles were damaged by each bubble. The floor is made up of a rectangular grid of 2 ft. by 2 ft. tiles. A bubble damages a tile if the tile is touched or contained by the bubble.

The Input:

The first line will contain two integers (h, v) (1 <= h, v <= 50) specifying the size of the grid of floor tiles. The floor is a rectangular grid of h (horizontal) by v (vertical) tiles. On the second line, there will be a single integer n specifying the number of bubbles to test. On the following n lines, there will be a single bubble each. A bubble is specified as a circle (we assume it has already hit the floor and popped) with a center x, y and a radius r (r > 0), respectively. x, y and r are in feet and all bubbles will be fully contained within the room.

The Output:

For each bubble, output a header "Bubble #i:" where i is the ith bubble being tested. Under the header, list all the carpet tiles that were damaged by that bubble. A floor tile's location is represented by its "coordinates" in the grid. The origin is in the lower-left corner. For example, the lower-left tile is (1, 1) and the one to its right is (2, 1). Leave a blank line between output for each bubble.

Sample Input:

```
5 5
2
1.5 1.5 0.25
4.0 4.0 1.0
```

Sample Output:

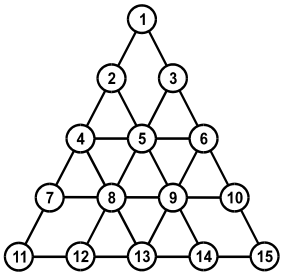
```
Bubble #1:
   Floor tile (1, 1) is damaged.

Bubble #2:
   Floor tile (2, 2) is damaged.
   Floor tile (3, 2) is damaged.
   Floor tile (2, 3) is damaged.
```

```
Floor tile (3, 3) is damaged.
```

Jump Puzzle

Filename: JUMP

You've all seen the "logic" puzzles which consist of a board with holes and pegs—you can jump one peg over another to remove the jumped-over peg. At the Apple Barrow restaurant there are several of these puzzles on the table, and in order to prove your intellect you've been asked to solve several of these puzzles. Not all of them are solved by leaving only one peg; one particularly interesting variation of the standard triangular puzzle is to leave eight pegs with no possible moves left. Being a programmer you quickly realize that it would be easier to write a program to solve a particular puzzle than to solve it through trial and error manually.



In general these boards can be quite large and complex, but the ones at the restaurant are small, very symmetric, and no peg has more than four moves. An example of this is the well-known, 15 hole, triangular puzzle shown to the right. You also know that all of the puzzles you've been asked to solve have solutions.

The Input:

There will be multiple data sets, each set has one board description and multiple puzzles. Input starts with an integer on a line by itself indicating the number of data sets. Each data set starts with an integer, *n*, which indicates the number of holes on the board. The following *n* lines each contain *n* integers separated by at least one space. Each line is a description of the moves possible from a hole. If there is a move from hole *a* over hole *b* to hole *c*, then the c^{th} integer on the a^{th} line will be *b*. If no move exists from *a* to *c*, then the c^{th} integer on the a^{th} line will be 0. For example, in the board shown above, the 13^{th} hole description will be "0 0 0 8 0 9 0 0 0 0 12 0 0 0 14" because you can move a peg in hole 13 to hole 4 by jumping over a peg in hole 8; you can move from hole 13 to hole 6 by going over hole 9; from 13 to 11 over 12; and from 13 to 15 over 14.

Following the description of the board will be the descriptions of the different puzzles for that board, starting with an integer *m* indicating the number of puzzles. Each puzzle consists of two lines of characters indicating the initial and solved configurations of the board. A configuration consists of *n* contiguous characters where a '`*`' indicates a hole with a peg and a '`.`' indicates an empty hole. For example, the typical initial configuration for the triangular board (all holes full except hole 1) would be, "`.**************`". You must find a sequence of moves from the initial configuration that result in the solved configuration. An example solved configuration would be "`*.............`" (one peg left in hole 1.)

The Output:

For each board, print a header (as shown in the sample.) Then process the puzzles for that board, outputting the solution as a sequence of moves (from hole -> to hole) . Follow the format shown in the sample output. If there is more than one way to solve a particular puzzle, print any one solution.

Sample Input:

```
1
15
0 0 0 2 0 3 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 4 0 5 0 0 0 0 0 0
0 0 0 0 0 0 0 5 0 6 0 0 0 0 0
2 0 0 0 0 5 0 0 0 0 7 0 8 0 0
0 0 0 0 0 0 0 0 0 0 0 8 0 9 0
3 0 0 5 0 0 0 0 0 0 0 0 9 0 10
0 4 0 0 0 0 0 0 8 0 0 0 0 0 0
0 0 5 0 0 0 0 0 0 9 0 0 0 0 0
0 5 0 0 0 0 8 0 0 0 0 0 0 0 0
0 0 6 0 0 0 0 9 0 0 0 0 0 0 0
0 0 0 7 0 0 0 0 0 0 0 0 12 0 0
0 0 0 0 8 0 0 0 0 0 0 0 0 13 0
0 0 0 8 0 9 0 0 0 0 12 0 0 0 14
0 0 0 0 9 0 0 0 0 0 0 13 0 0 0
0 0 0 0 0 10 0 0 0 0 0 0 14 0 0
1
.**************
*..............
```

Sample Output:

```
Board #1:
  Solution for puzzle #1:
     4 ->  1
     6 ->  4
     1 ->  6
     7 ->  2
    10 ->  3
    12 ->  5
    14 -> 12
    11 -> 13
    13 ->  6
     3 -> 10
    15 ->  6
     6 ->  4
     4 ->  1
```

Jason and His Friends

Filename: `FRIENDS`

Jason and his friends were over his house one day playing networked computer games. One of their favorites are first-person, shooter-type games. Unfortunately, these games only keep track of scores within a single game or round. Jason and his friends would like to know who is the best overall for the entire day and need your help.

The Problem:

Given the individual round scores for a player, compute his overall score.

The Input:

The first line of the input will contain two single integer p and r representing the number of players and the number of rounds, respectively. On the next p lines will be the scores for each of the players, respectively (r per line). A positive score means that the player had more kills than losses that round; a negative score means that the player had more losses than kills that round.

The Output:

For each player, compute his overall score and output it on a separate line.

Sample Input:

```
3 5
4 3 1 -3 7
9 0 -3 -5 2
4 3 -1 3 5
```

Sample Output:

```
12
3
14
```

Dictionary Sorting

Filename: SORT

Some languages have prefixes which count as a letter when sorting. For example, in Spanish 'ch' comes after 'c' and before 'd'. So "curso" comes before "chalar" comes before "dar" in the dictionary; in the 'C', 'Ch' and 'D' sections, respectively. You are to write a program that will arrange text into dictionary order for arbitrary languages.

A language will be specified as a listing of prefixes that are used to sort the text. Prefixes will be unique, but may share starting letters. A word can only be in one section of the dictionary, so only the longest matching prefix is used.

The Input:

There will be multiple data sets. Each set starts with an integer n ($1 \le n \le 100$) specifying the number of prefixes. The prefixes will be on the next n lines in the proper sort order. Prefixes will contain only lower case letters, will not be longer than 20 characters, and there will be no extra whitespace.

Following the prefixes will be an integer m ($1 \le m \le 1000$) specifying the number of words to sort. The words will be on the next m lines. Words will contain only lower case letters and will be no longer than 20 characters. Words will not be repeated within a single data set. Again, there will be no extra whitespace.

The word(s) will start with at least one of the prefixes.

End of input is indicated by $n = 0$.

The Output:

For each input set print the set number (see the sample output.) Then, for each prefix that has words in it, print the heading (capitalize the first letter) on a line by itself. Then print the words with that prefix in alphabetical order, indenting each one two spaces. Output for the prefixes should be in the order the prefixes appear in the input and leave a blank line after each prefix's words.

Sample Input:

```
8
a
b
c
ch
d
x
y
z
3
```

```
curso
chalar
dar
0
```

Sample Output:

```
---Dictionary #1---
C
  curso

Ch
  chalar

D
  dar
```