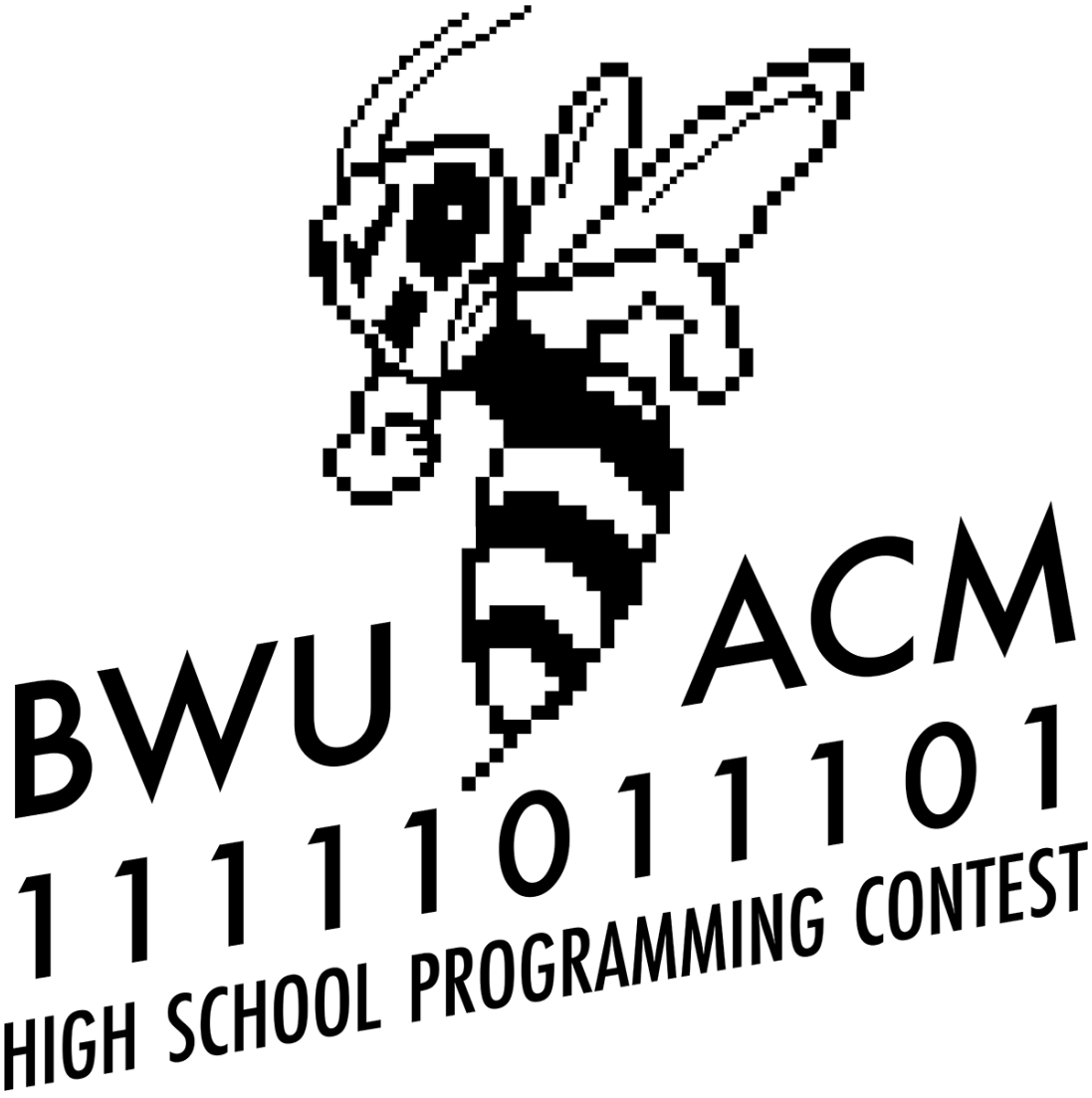
BALDWIN WALLACE UNIVERSITY
2013 PROGRAMMING CONTEST



DO NOT OPEN UNTIL INSTRUCTED TO DO SO!

Mystery Message

Marvin the Paranoid Android needs to send an encrypted message to Arthur Den. Marvin is absurdly smart and wants to use a super sophisticated rotation cipher. Unfortunately, the job of writing the program to encode the message is beneath him, so now it's your job. Messages will contain any combination of lower case letters 'a' through 'z'. There will be no spaces, punctuation or special characters.

A rotation cipher works by replacing each character in the message with one that is a specified shift (i.e., number of characters) away from it in the alphabet. For example, if the shift is 3, then 'a' is replaced by the character three positions later in the alphabet (i.e., 'd'). It is possible that the size of the shift would go beyond the end of the alphabet. In these cases, the shift wraps around to the beginning of the alphabet. For example, if the shift were 3 and the character to be replaced were 'y', 'y' is replaced by 'b', which is three characters away (one at the end of the alphabet, then two from the start).

Details of the input

The first line of input will be an integer, *n*, indicating the number of cases to be tested. Each of the following *n* lines will contain an integer, *s*, the shift ($0 < s < 1{,}000{,}000$), followed by a single space, and then the message that you are to encode.

Details of the output

Each output line should begin with "Case *i:*" where *i* is the appropriate test case, followed by a single space, then the encrypted string.

Sample input

```
2
3 abc
7 za
```

Sample output

```
Case 1: def
Case 2: gh
```

Road Trip

The Smiths want to take a family trip down to IKEA. They live in Cleveland, OH, and the nearest IKEA is in Pittsburgh, PA, which is 140 miles away. Since gas is expensive, they want to make sure they are using the cheapest way possible to get there. They own 3 different cars: Car A gets 11 miles per gallon; Car B gets 25 miles per gallon; and Car C gets 35 miles per gallon. They also have an Uncle who owns a Prius that gets 49 miles per gallon. The uncle allows them to borrow his Prius, even when he is on the road for work, (since it just sits in some parking lot all day otherwise). However, if they want to borrow his Prius, they must first drive one of their cars from their home to his current location, use the Prius to get from that location to IKEA, drive the Prius back to the uncle, and then use their own car to get home. Your program is to determine what travel option is the most economical.

Details of the input

The first line of input will contain a single integer *n* which will represent the number of test cases to test. Each of the following *n* lines will contain three real numbers, *c* (the cost of a gallon of gas), *f* (the uncle's current distance from the family, in miles), and *p* (the uncle's current distance from IKEA, in miles), separated by a single space.

Details of the output

For each test case, output should begin with the line "Case *i*:" where *i* is the corresponding case number. That should be followed by a single space, and then the minimum roundtrip cost to get the Smith family from home, to IKEA, and back home again. The minimum cost should be truncated to two decimal places and preceded by a $.

Sample input

```
2
3.50 35 98
4.00 52.5 147
```

Sample output

```
Case 1: $21.00
Case 2: $32.00
```

Football Frenzy

Fred has been invited to a Football Party (American Football that is), but is afraid because he doesn't know very much about the sport. After a quick internet search, he felt a little more comfortable, and was somewhat intrigued by the various ways to score points. He hopes to impress the other partygoers by telling them how many different ways a team can get a certain score. He doesn't care about the order the points are received. As far as he is concerned, a touchdown followed by a field goal is the same as a field goal followed by a touchdown. Thus, for instance, there is only one way to get a score of 3 (a field goal), but two ways to get 7 (a field goal and two safeties, or a touchdown and an extra point).

To help him avoid making a fool of himself at the party by miscalculating, you agree to write a program that determines the number of ways to attain a given total score. His research has uncovered the following information on ways to score points in a football game:

Field goal:	3 points
Safety:	2 points
Touchdown:	6 points -- but, with each touchdown, there can be a kick for an extra point, a 2-point conversion, or neither - adding 1, 2, or 0 additional points respectively.

A team can score any number of field goals, safeties, and touchdowns. However, extra points and 2-point conversions can only occur following a touchdown, and only one of them can be attempted after each touchdown.

Details of the input

The first line of input will contain a single integer, *n*, indicating the number of football scores. Each of the following *n* lines will contain an integer *k* ($0 < k < 100$), a football team score.

Details of the output

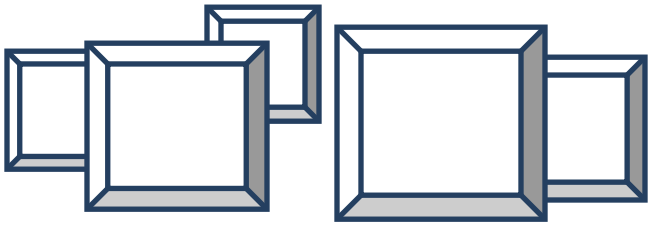
For each test case, output should begin with the line "Case *i*:" where *i* is the corresponding case number. That should be followed by a single space and the number of ways to get the given score.

Sample input

```
5
3
7
14
8
5
```

Sample output

```
Case 1: 1
Case 2: 2
Case 3: 11
Case 4: 4
Case 5: 1
```



Beeblebrox's Picture Frames

Zaphod Beeblebrox is so taken with himself that he has decided to display copies of his picture everywhere. To enhance the pictures and draw attention to the fact that he is a star (or at least believes he is), he wants each picture to be framed within a border of asterisks. However, given the number of pictures he'd like to display, the chore of producing a border for each is more than he cares to do. Thus, he would like a program to automatically produce a border for each picture.

Each of his pictures has equal width and length … i.e., is square. The framed picture is to be of the same size as the original, with the border having a thickness of a specified size. That is, the border covers the outside edges of the picture leaving the room in the center (at least one space) to display Zaphod's image. If it is not possible to print a border of the given width within the given size while leaving space for his picture, the program should print "BAD DIMENSIONS".

Details of the Input

The first line of input will be an integer, *n*, indicating the number of cases to be tested. Each of the following *n* lines will contain two integers, *s* and *b*, separated by a single space, where *s* indicates the dimension of the finished, bordered picture and *b* is the width of the border. Dimensions are given in terms of number of characters.

Details of the Output

For each test case, output should begin with the line "Case *i*:" where *i* is the corresponding case number. If a border is possible for that case, it should be printed, beginning on the next line; if not, "BAD DIMENSIONS" should be printed on the next line.

Sample input

```
2
4 1
1 4
```

Sample output

```
Case 1:
****
*  *
*  *
****
Case 2:
BAD DIMENSIONS
```

Alien Chess

You've been taken hostage by aliens. However, they're relatively fair aliens. They have given you a chance to win your freedom by winning what they believe to be how the Earthlings play Chess. However, their version of Chess has a few key differences:

1. Every move must start with your Super Pawn taking a piece.
2. A Super Pawn behaves like the piece it just took. At the start of the game it acts as a pawn.
3. You may perform a combo by, after taking a piece with your Super Pawn, taking another piece with your Super Pawn.
4. You win by capturing the Enemy's Super Pawn.
5. The only piece on the board that belongs to you is your Super Pawn. The remaining pieces belong to your Enemy and include the Enemy's Super Pawn, regular pawns, rooks, bishops, and knights.
6. There can be any number of rooks, pawns, and knights, but only one Super Pawn and one Enemy Super Pawn.

A pawn can only capture a piece by moving one space diagonally forward. Thus the pawn P can only take pieces in the spots marked with an X. Forward is always oriented in this direction for you.

```
_ _ _ _ _ _ _ _
_ _ _ X _ X _ _
_ _ _ _ P _ _ _
_ _ _ _ _ _ _ _
```

A rook can capture any piece located directly above, below, or to either side of it as long as no other piece is between the rook and that piece.

A bishop can capture any piece on the diagonals from itself as long as no other piece is between the bishop and that piece.

A knight can capture pieces in any of the 8 squares located by moving two squares horizontally and one square vertically, or two squares vertically and one square horizontally. So a knight, marked by K, can capture any piece in the spots marked with an X.
NOTE: A knight 'jumps' over any other piece between itself and its target. In other words, it cannot be blocked.

```
_ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _
_ _ X _ X _ _ _
_ X _ _ _ X _ _
_ _ _ K _ _ _ _
_ X _ _ _ X _ _
_ _ X _ X _ _ _
_ _ _ _ _ _ _ _
```

You are currently on the last possible move before the aliens get bored of the game and revoke their offer to free you. You are to determine whether or not your next move will free you.

Details of the Input

The first line is a single integer, *n,* denoting the number of games to be played. For each game, the board is described by 8 lines of 8 characters each. The only possible characters are:

- '-' an empty position
- 'S' your Super Pawn, which is also your starting position
- 'E' your enemy's Super Pawn, which is your ending position
- 'P' a Pawn
- 'R' a Rook
- 'B' a Bishop
- 'K' a Knight

There will be one 'S' and one 'E' on each board. There will be one blank line separating one board description from the next.

Details of the Output

There should be one line of output for each of the *n* boards. Each output line should begin with "Case *i*:" where *i* is the corresponding case number. That should be followed by a single space, then either "Yes" if you can win, or "No" if you cannot win on our next turn.

Sample Input

```
3
-------E
------P-
-----P--
----P---
---P----
--P-----
-P------
S-------

----E---
--R---R-
--------
--------
--------
--R---R-
-P------
S-------

----R--E
--------
---K----
--------
--------
------B-
-----S--
--------
```

Sample Output

```
Case 1: Yes
Case 2: No
Case 3: Yes
```

Prime Time

Every non-prime number can be expressed as a product of two or more prime numbers. The process of doing so is called prime factorization. When a number is prime, its only factor is itself. For example, the number 1421 is not prime since it can be written as 7x7x29, whereas 5 is prime since it can only be written as 5. Your job will be to determine the unique prime factors for a given number and print them in descending order.

Details of the Input

The first line of input will contain a single integer, *n*, which indicates the number of test cases to consider. Each of the following *n* lines will contain a single positive integer *y* ($1 < y \leq 10000$) whose prime factors you are to find.

Details of the Output

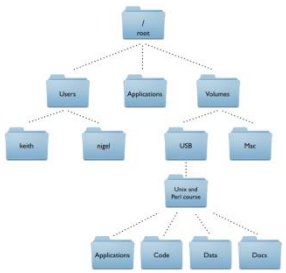
For each test case, output should begin with the line "Case *i*:" where *i* is the corresponding case number. That should be followed by a single space, then the given number's unique prime factors in descending order separated by a single space.

Sample Input

```
4
256
1421
9999
17
```

Sample Output

```
Case 1: 2
Case 2: 29 7
Case 3: 101 11 3
Case 4: 17
```



Where is My /water?

Your boss, a.k.a. Lazy Larry, is always trying to find the simplest way to get every task done. For the past week, he has been complaining about how much effort it takes to move between file folders on his computer. So, he's made it your job to write him a utility program that will determine the least number of mouse clicks he should need to get from his current directory to another directory of interest.

You realize that in the simplest case, a graphical representation of the directory structure would be a simple tree like that pictured above. Given two directory paths, e.g. C:/this/is/where/you/start/ and C:/where/is/my/water/, the problem could be solved by finding the longest partial path shared by the files (in this case C:/) and then counting the number of back clicks to that point from the starting folder (5) and forward clicks down to the destination (4) to get the total number required (9).

But, Lazy Larry, being who he is, has complicated the problem by putting shortcuts within some directories that, if used, might make getting to the destination directory possible in fewer clicks. For example, if there is a shortcut [C:/this/is/] in the start folder, then it is shorter to take one click into the start folder, follow the shortcut (saving one click), and then proceeding as above for a total of 8 clicks.

Details of the input

Folder names will be complete paths starting from the root of the C:/ drive. That is,

C:/*s1*/*s2*/…/*sx*/

where each of *s1*, *s2*, ..., *sx* is the name of a subfolder. A subfolder name is a simple alphanumeric string (i.e., no special characters) followed by a slash (/). The final of these names, *sx/*, may be replaced by the name of a shortcut to another folder. A shortcut will appear as a complete path enclosed in square brackets. That is, [C:/*s1*/*s2*/…/*sx*/]. Shortcuts may not contain other shortcuts.

Input to the program will be as follows:

The first line is a single integer, *n*, which specifies the number of folder paths that follow.
Each of the next *n* lines will contain a single string, *p*, which specifies a path to a folder or shortcut.
The next line is a single integer, *m*, which specifies the number of test cases that will follow.
Each of the next *m* lines will contain a single string, *d*, which specifies a starting folder.

There will be no input for the destination folder of each case. It is defined to be C:/where/is/my/water/.

Details of the output

For each test case, output should begin with the line "Case *i*:" where *i* is the corresponding case number. That should be followed by a single space and the minimum number of clicks for that case.

Sample input

```
11
C:/
C:/where/
C:/where /is/
C:/where/is/my/
C:/where/is/my/water/
C:/this/
C:/this/is/
C:/this/is/where/
C:/this/is/where/you/
C:/this/is/where/you/start/
C:/this/is/where/you/start/[C:/this/is/]
2
C:/this/
C:/where/is/my/water/
```

Sample output

```
Case 1: 5
Case 2: 0
```