# Problem A: Arithmetically Challenged

Challenge 24 is a popular mathematics game used in many grade schools. In each game, contestants are given a card with four positive integers $i_1$, $i_2$, $i_3$, $i_4$ on it, and the first one who can use all of these numbers and any combination of the four basic arithmetic operations to get 24 wins. Each of the numbers $i_1$, $i_2$, $i_3$, $i_4$ must be used exactly once. Division can be used only if the divisor evenly divides the dividend (i.e., you can perform 6/2 but not 6/4). For example, if the card contains the numbers 7, 2, 5 and 1, possible solutions are (7-2)*5-1 or (7+1)*(5-2). Hmmm . . . this sounds like a source of a good programming problem.

Write a program that determines the longest consecutive sequence of integers that can be obtained by different ways of arithmetically combining the four integers. For example, with 7, 2, 5 and 1 the longest consecutive sequence is -18 to 26 (yes, we're allowing final results to be negative). The "+" and "−" operators must be used as binary operators, not as unary signs.

## Input

Each test case will consist of a single line containing the four, not necessarily distinct, positive integers, none of which will exceed 100. A line containing four 0's will terminate input.

## Output

For each test case, output the case number and the longest consecutive sequence of obtainable values, in the format shown in the sample output. If there is more than one longest consecutive sequence, use the one with the largest first value.

## Sample Input

```
7 2 5 1
8 15 38 3
0 0 0 0
```

## Sample Output

```
Case 1: -18 to 26
Case 2: 150 to 153
```

A

2009 ACM ICPC South Central USA Regional Programming Contest

# Phil In the Blanks

**Introduction:**

Phil is trying to solve a series of puzzles in which he must fill in blanks in a sentence with number words to make true statements. Being a programmer, you think you can write a program to find possible solutions. An example of a puzzle:

**There are ___ Os and ___ Ts in this sentence.**

One possible solution would be to put "two" in the first blank and "five" in the second blank. Note that "one" would not work in the first blank because the word "one" has one "O" in it; i.e., the words put in the blanks are part of the puzzle. Also note that grammar is not considered.

**Input:**

Input to this problem will begin with a line containing a single integer $D$ ($1 \le D \le 100$) indicating the number of data sets. Each data set will consist of a single line, which will be the puzzle to solve. Each puzzle will be 1–100 characters (inclusive) and will contain 1-4 positive assertions (inclusive) of the following variety:

| | |
|---|---|
| ___ *C*s | the blank should be filled in with the number of all instances (upper and lowercase) of the indicated letter *C* in the puzzle |
| ___ `letters` | the blank should be filled in with the number of letters in the puzzle |
| ___ `vowels` | the blank should be filled in with the number of vowels in the puzzle; for the purposes of this problem only the letters A, E, I, O, or U are considered to be vowels |
| ___ `consonants` | the blank should be filled in with the number of consonants in the puzzle |

Note that the blanks above consist of three underscore characters ("_") in a row, and that the literal words "letters", "vowels", and "consonants" are always in lowercase. Any underscore characters appearing in the puzzles will be part of a blank.

**Output:**

For each data set output the number of possible correct solutions. All number words used in the puzzles will be from "zero" to "one hundred".

**Sample Input:**

```
5
There are ___ Os and ___ Ts in this sentence.
There are ___ Ts and ___ Es in this sentence.
Hey, ___ letters here
```

```
And ___ vowels here
with ___ consonants here
```

**Sample Output:**

```
1
0
2
2
2
```

C

# Problem G: Shuffling

**Source: `shuffling.{c,cpp,java}`**
**Input: `console {stdin,cin,System.in}`**
**Output: `console {stdout,cout,System.out}`**

A casino owns an expensive card shuffling machine which may shuffle up to 520 cards at a time (there are 52 cards in each deck). For convenience, we will simply label the cards 1, 2, 3, ..., **`N`** where **`N`** is the total number of cards, and copies of the same card (e.g. Ace of Spades) from different decks are considered different. Unfortunately, the card shuffling machine is defective, and it always shuffles the cards the same way. The company that produces these machines is out of business because of the economic downturn. There is no one who can fix the machine, and a new machine is too expensive.

Being a brilliant employee of the casino, you realized that all is not lost. You can shuffle the cards differently simply by using the machine zero or more times. For example, suppose that the machine shuffles the cards 1, 2, 3, 4 into the order 2, 3, 4, 1. If you put the cards into the machine, take the shuffled cards out and insert them into the machine again (without changing the order), you will get the order 3, 4, 1, 2. That way, it is possible to shuffle the cards in many different ways even though it may take longer. But this is not a significant issue since decks do not have to be reshuffled often, and used decks can be shuffled while other decks are being used to avoid any waiting time.

Unfortunately, not all shufflings can be produced in this way in general, and you wish to know if this procedure "stack the decks" in a favorable way for the casino or the player. As a first step, you wish to know which shufflings are possible to produce, and how many times you need to use the machine on the deck in order to produce the shuffling.

## Input

The input for each case consists of three lines. The first line consists of a single integer **N** indicating the number of cards to shuffle. The number of cards is a positive integer up to 520. The second line consists of the integers 1, 2, ..., **N** listed in some order and separated by a space. The list gives the order of the shuffling performed by the machine when the input cards are ordered 1, 2, ..., **N**. The third line is in the same format as the second line, and gives the shuffling we wish to obtain. The end of input is indicated by a line in which **N** = 0.

## Output

For each case, print the smallest number of times (zero or more) you need to pass the deck through the machine to produce the desired shuffling. If it is not possible, print -1. The output for each case should be in a single line. You may assume that the answer will always fit in a 32-bit signed integer.

## Sample Input

```
4
2 3 4 1
3 4 1 2
4
2 3 4 1
1 3 2 4
10
2 1 3 5 6 7 8 9 10 4
1 2 3 9 10 4 5 6 7 8
0
```

## Sample Output

```
2
-1
12
```

# Problem B: Cover Up

"The Price is Right" is a popular game show where contestants play various games to win fabulous prizes. One of the games played on the show is called "Cover Up" whose object is to guess a 5-digit number (actually, the price of a new car). In the actual game, contestants are given two numbers to choose from for the first digit, three numbers to choose from for the second digit, and so on. A contestant selects one number for each digit (from the set of yet unpicked numbers for that digit) and then is told which ones are correct; if at least one is correct, the player is allowed to guess again for all incorrect digits. The contestant keeps guessing as long as they keep getting at least one new digit correct. The game ends when either all the digits are correct (a win for the contestant) or after a turn when no new digit is guessed correctly (a loss).

Typically this game is not sheer luck. For example, suppose you had the following five possibilities for the last digit: 1, 3, 5, 8 and 9. Many car prices end with either a 5 or a 9, so you might have, say, a 70% chance that one of these two numbers is correct; this breaks down to a 35% chance for either the 5 or the 9 and a 10% chance for each of the other three digits. Now say you pick the 5 and it's wrong, but some other guess you made was right so you still get to play. With this additional information the probabilities for the remaining 4 numbers change: the probability for the 9 is now close to around 54%, while each of the other three numbers now has a little over a 15% chance. (We'll let you figure out how we got these values). We'll call the 5 and the 9 in the original group the *known* candidates, and typically there are known candidates in other columns as well. For example, if the two numbers for the first (high order) digit are 1 and 9, the contestant can be 100% sure that the 1 is the correct digit (there aren't too many $90,000 cars to be given away).

For this problem, you are to determine the probability of winning the game if an optimal strategy for picking the numbers (based on probabilities such as those described above) is used.

## Input

Each test case will consist of two lines. The first will be $n$, the number of digits in the number to be guessed. The maximum value of $n$ will be 5. The second line will contain $n$ triplets of numbers of the form $m$ $l$ $p$ where $m$ is the number of choices for a digit, $l$ is the number of known candidates, and $p$ is the probability that one of the known candidates is correct. In all cases $0 \leq l < m \leq 10$ and $0.0 \leq p \leq 1.0$. Whenever $l = 0$ (i.e., when there are no known candidates) $p$ will always be 0.0. A line containing a single 0 will terminate the input.

## Output

Output for each test case is the probability of winning using optimal strategy. All probabilities should be rounded to the nearest thousandth, and trailing 0's should not be output. (A 100% chance of winning should be output as 1.)

### Sample Input

```
2
3 1 0.8 2 0 0.0
2
3 2 0.8 2 0 0.0
2
3 2 0.82 2 1 0.57
3
4 1 1.0 3 0 0.0 10 1 1.0
0
```

### Sample Output

```
0.85
0.6
0.644
1
```

2009 ACM ICPC South Central USA Regional Programming Contest

# YO!

**Introduction:**

While sitting in traffic on I-35 one day, you look up at the pickup in front of you and notice the word "YO" staring back at you from the tailgate. After a bit of closer investigation, you realize that the witty driver of the truck is driving a Toyota and has simply painted over the "TO" and "YA". As the traffic continues to sit, you beging to daydream and wonder how well this technique would work for other words.

Given a starting word and a dictionary, determine how many ways you could "paint over" combinations of letters in the starting word and end up with one or more correctly spelled words. If a "paint over" combination produces multiple words (including the same word appearing at multiple locations in the starting word), then it's counted separately from the combinations that produce individual words. Multiple words need not be separated, and a single word may contain embedded spaces. For example, if the dictionary contains "he" and "vet" and the starting word is "CHEVROLET", then four possible combinations can be produced:

```
.HE......
.H.....E.
...V...ET
.HEV...ET
```

However, multiple words may not overlap each other in any way. For example, the starting word "CHEVROLET" can form individual words "hoe" and "vet", but the combination of both words would not be valid.

**Input:**

The input will begin with a line containing a single integer $N$ ($1 \le N \le 100$) indicating the number of data sets. Each data set will begin with a line of the form "$X$ $S$". The integer $X$ ($1 \le X \le 200000$) indicates the number of words in the dictionary for this data set. The string $S$ specifies the starting word and will contain at least 1 character but no more than 30. The next $X$ lines will be the dictionary for the data set and will each contain one word that is also at least 1 and at most 30 characters long.

Note that the starting word will always be in upper case, while the words in the dictionary may be mixed case. Neither the starting word nor the words in the dictionary will contain any embedded spaces, and the dictionary will not contain any duplicate words. And no, the dictionary is not guaranteed to be in any particular order.

**Output:**

For each data set in the input, output a line containing a single integer representing how many ways you could "paint over" combinations of letters in the starting word and end up with one or more correctly spelled words. At least one letter in the starting word has to be painted over to count; if the dictionary contains the same exact word as the starting word then, then this word would *not* be counted. You may assume that the total number of combinations per data set will not exceed 300000.

**Sample Input:**

```
2
2 TOYOTA
toy
yo
5 CHEVROLET
does
he
like
her
program
```

**Sample Output:**

```
2
3
```

# Problem D: Flipper

Little Bobby Roberts (son of Big Bob, of Problem G) plays this solitaire memory game called Flipper. He starts with $n$ cards, numbered 1 through $n$, and lays them out in a row with the cards in order left-to-right. (Card 1 is on the far left; card $n$ is on the far right.) Some cards are face up and some are face down. Bobby then performs $n-1$ flips — either right flips or left flips. In a right flip he takes the pile to the far right and flips it over onto the card to its immediate left. For example, if the rightmost pile has cards A, B, C (from top to bottom) and card D is to the immediate left, then flipping the pile over onto card D would result in a pile of 4 cards: C, B, A, D (from top to bottom). A left flip is analogous.

The very last flip performed will result in one pile of cards — some face up, some face down. For example, suppose Bobby deals out 5 cards (numbered 1 through 5) with cards 1 through 3 initially face up and cards 4 and 5 initially face down. If Bobby performs 2 right flips, then 2 left flips, the pile will be (from top to bottom) a face down 2, a face up 1, a face up 4, a face down 5, and a face up 3.

Now Bobby is very sharp and you can ask him what card is in any position and he can tell you!!! You will write a program that matches Bobby's amazing feat.

## Input

Each test case will consist of 4 lines. The first line will be a positive integer $n$ ($2 \leq n \leq 100$) which is the number of cards laid out. The second line will be a string of $n$ characters. A character `U` indicates the corresponding card is dealt face up and a character `D` indicates the card is face down. The third line is a string of $n-1$ characters indicating the order of the flips Bobby performs. Each character is either `R`, indicating a right flip, or `L`, indicating a left flip. The fourth line is of the form $m\ q_1\ q_2\ \ldots\ q_m$, where $m$ is a positive integer and $1 \leq q_i \leq n$. Each $q_i$ is a query on a position of a card in the pile (1 being the top card, $n$ being the bottom card). A line containing 0 indicates end of input.

## Output

Each test case should generate $m+1$ lines of output. The first line is of the form

`Pile` $t$

where $t$ is the number of the test case (starting at 1). Each of the next $m$ lines should be of the form

`Card` $q_i$ `is a face up` $k$.

or

`Card` $q_i$ `is a face down` $k$.

accordingly, for $i = 1, .., m$, where $k$ is the number of the card.

For instance, in the above example with 5 cards, if $q_i = 3$, then the answer would be

```
Card 3 is a face up 4.
```

### Sample Input

```
5
UUUDD
RRLL
5 1 2 3 4 5
10
UUDDUUDDUU
LLLRRRLRL
4 3 7 6 1
0
```

### Sample Output

```
Pile 1
Card 1 is a face down 2.
Card 2 is a face up 1.
Card 3 is a face up 4.
Card 4 is a face down 5.
Card 5 is a face up 3.
Pile 2
Card 3 is a face down 1.
Card 7 is a face down 9.
Card 6 is a face up 7.
Card 1 is a face down 5.
```

acm International Collegiate Programming Contest
IBM event sponsor
2009 ACM ICPC South Central USA Regional Programming Contest

# I'm Attacking the Darkness!

**Introduction:**

Many tabletop role-playing games (RPGs), such as a rather famous one involving dragons and dungeons, require the use of dice to simulate various random events in the game. Unlike most regular board games that use six-sided dice, RPGs typically require the use of several polyhedral dice with 4, 6, 8, 12, or 20 sides. Often times the player is required to make a skill check. This involves rolling a set of dice, adding the individual dice rolls together, and then comparing the sum against a pre-determined *target value*. If the total dice roll is equal to or greater than the target value, then the skill check is considered successful.

When a RPG requires a roll of the dice, it will use a *dice notation* to indicate how many and what kind of dice to roll. For example, "1d4+2d8" would direct the player to roll three dice, one 4-sided die plus two 8-sided ones, and then to add the results of all three dice rolls together. Sometimes the player is required to add or subtract a modifier (i.e. a constant integer) to the total sum of the dice roll. Using the dice notation, a +3 modifier (i.e. a bonus) might be specified as "1d4+2d8+3", while a negative modifier like −5 (i.e. a penalty) might be specified as "1d4−5+2d8". Note that the faces of each die are numbered starting with 1 and going up to the number of sides, so that every face has a unique number. For example, the faces on a 4 sided die would be numbered 1, 2, 3, and 4.

Because failing a skill check usually results in something bad happening to the player's character, it's important to know what the chances are of a particular skill check succeeding. To help in this regard, you decided to write a program that reads in a target value along with a dice roll expression, and then prints the probability of passing that skill check.

**Input:**

Input to this problem will begin with a line containing a single integer $N$ ($1 \le N \le 100$) indicating the number of data sets. Each data set consists of a single line of the form "$T$ $X$", where $T$ ($0 \le T \le 100$) is the target value, and $X$ is a string containing the dice notation. $X$ will not contain any spaces.

The dice notation $X$ will contain one or more *terms* separated by either a plus "+" or minus "−" sign. Each term can be either an integer $M$ ($1 \le M \le 10$) or an expression of the form "$N$d$S$" where $N$ ($1 \le N \le 6$) and $S$ is an integer equal to 4, 6, 8, 12, or 20. The total number of dice to roll for any given expression will not exceed 6.

**Output:**

For each data set, print a single line with the probability that the given dice roll will be equal to or higher than the target value $T$. If the target value cannot be achieved, then print the number zero (0). If the target value can always be achieved with every possible dice roll (e.g. due to modifiers), then print the number one (1). Otherwise, print the probability as a fraction reduced to its lowest terms.

**Sample Input:**

```
2
7 1d6
7 2d4+1
```

**Sample Output:**

```
0
3/8
```

## Problem F: Here's a Product Which Will Make You Tensor

Most people are familiar with how to multiply two matrices together. However, an alternate form of multiplication known as tensor multiplication exists as well, and works more like you would expect matrix multiplication should. Let $A$ be a $p \times q$ matrix and $B$ be an $n \times m$ matrix, where neither $A$ nor $B$ is a $1 \times 1$ matrix. Then the tensor product $A \otimes B$ is a $pn \times qm$ matrix formed by replacing each element $a_{ij}$ in A with the matrix $(a_{ij}) \cdot B$. Two examples are shown below, which also demonstrate that, like normal matrix multiplication, tensor multiplication is non-commutative:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 2 & 2 & 2 \\ 1 & 1 & 1 & 2 & 2 & 2 \\ 1 & 1 & 2 & 2 & 2 & 4 \\ 3 & 3 & 3 & 4 & 4 & 4 \\ 3 & 3 & 3 & 4 & 4 & 4 \\ 3 & 3 & 6 & 4 & 4 & 8 \end{bmatrix}, \qquad \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 2 \end{bmatrix} \otimes \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \\ 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \\ 1 & 2 & 1 & 2 & 2 & 4 \\ 3 & 4 & 3 & 4 & 6 & 8 \end{bmatrix}$$

Note that there is no restriction that the number of columns in the first matrix must equal the number of rows in the second, as there is with normal matrix multiplication. The object of this problem is to determine the number of ways (if any) a given matrix can be formed as a result of a tensor multiplication.

### Input

The first line of input for a test case will contain two positive integers $r$ and $c$ indicating the number of rows and columns in the matrix. After this will follow $r$ lines each containing $c$ positive integers. The values of $r$ and $c$ will be $\leq 500$, each entry in the matrix will be no greater than 65,536, and the last test case is followed by a line containing `0 0`.

### Output

For each test case, output the number of different ways the matrix could be the tensor product of two positive integer matrices, neither of which is a $1 \times 1$ matrix.

### Sample Input

```
6 6
1 1 1 2 2 2
1 1 1 2 2 2
1 1 2 2 2 4
3 3 3 4 4 4
3 3 3 4 4 4
3 3 6 4 4 8
2 2
3 6
4 9
2 4
15 18 30 36
20 24 40 48
0 0
```

### Sample Output

```
1
0
4
```

H

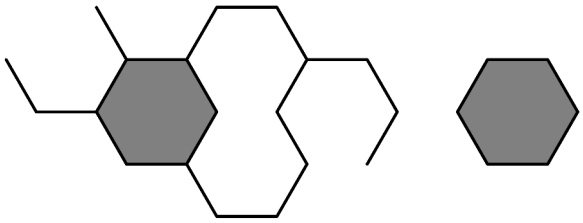
2009 ACM ICPC South Central USA Regional Programming Contest

# Honeycomb, Honeycomb, Me Want Honeycomb!

**Introduction:**

A windstorm has knocked over a beekeeper's hive boxes. A hive box contains a number of panels. Each panel contains a honeycomb. The panels are thin enough that the bees create a single layer of hexagonal cells in each panel. All cells are hexagonal and of uniform size.

Upon inspecting the panels, the beekeeper discovers that many of the hexagonal cells have been damaged. Given as input a list of line segments representing undamaged cell walls, compute the number of undamaged (i.e. containing all six walls) hexagons in the panel. You can assume that the only damage to the honeycomb is that some cell walls are missing, but none of them are moved, broken in half, etc. Below is an example honeycomb with the undamaged hexagons shaded in gray:



**Input:**

Input to this problem will begin with a line containing a single integer $N$ ($1 \le N \le 100$) indicating the number of data sets. Each data set begins with a line containing a single integer $S$ ($1 \le S \le 1000$) specifying the number of line segments in the data set. This is followed by $S$ lines of the form "$X_1, Y_1\ \ X_2, Y_2$" which specify the individual cell walls of the honeycomb. Each coordinate is a floating point number greater than or equal to zero but less than or equal to 1000 and with at most 3 digits after the decimal point (i.e. rounded to the nearest thousandth). The coordinates will *not* use "exponent notation" such as "`3.123e+3`".

You can make the following assumptions about the input:

- The length of each cell wall is 1 unit
- The honeycomb will have the same orientation as the example figure. In other words, if the top or bottom cell walls of a unit are present, they will be always parellel to the x-axis.
- There will be no duplicate or overlapping cell walls. The line segments can only touch each other at the endpoints.

**Output:**

For each data set, print the number of undamaged hexagonal cells that were detected.

**Sample Input:**

```
6
0.500,1.866 1.500,1.866
0.000,1.000 0.500,1.866
1.500,0.134 2.000,1.000
1.500,1.866 2.000,1.000
0.500,0.134 1.500,0.134
0.000,1.000 0.500,0.134
3
1.500,1.866 2.000,1.000
0.500,0.134 1.500,0.134
0.000,1.000 0.500,0.134
```

**Sample Output:**

```
1
0
```

## Problem G: Trip the Lights Fantastic

Bob Roberts (father of Little Bobby of problem D) works at the Traffic Commission for a medium size town. Bob is in charge of monitoring the traffic lights in the city and dispatching repair crews when necessary. Needless to say, Bob has a lot of free time, so to while away the hours he tries to figure out the quickest way to take short trips between various points in the city. Bob has at his disposal a lot of information: the layout of streets in the city and the location and cycle times for all of the traffic lights. To simplify the solution process, he makes the following assumptions:

1. All cars travel at the same top speed, and, if sitting at a red light, take 5 seconds to react and get up to speed. (That is, Bob assumes the car is essentially standing still for 5 seconds, then proceeds at top speed. Bob also assumes the light will not have turned back to red in the 5 seconds it takes to get going.)
2. Each car approaches a light at full speed and either passes through the light if it is green or yellow, or comes to an immediate stop if it is red. Cars are allowed to pass through a light if they hit it just as it is turning to green. Cars must stop if they reach the light just as it is turning to red.
3. The time to make turns through a light is ignored. It is possible to travel between any two lights, although perhaps not directly.

Furthermore, no u-turns are allowed nor will routes revisit an intersection. Even given these assumptions, Bob has difficulty coming up with minimum time paths. Let's see if you can help him.

### Input

The first line of each test case will contain four positive integers $n$, $m$, $s$, and $e$, where $n$ ($2 \leq n \leq 100$) is the number of traffic lights (numbered 0 through $n-1$), $m$ is the number of roads between the traffic lights, and $s$ and $e$ ($s \neq e$) are the starting and ending lights for the desired trip. There will then follow $n$ lines of the form $g$ $y$ $r$ indicating the number of seconds that each light is green, then yellow, then red. ($1 \leq g, y, r \leq 100$.) The first of these lines refers to light 0, the second to light 1, and so on. Following these $n$ lines will be $m$ lines, each describing one road. These lines will have the form $l1$ $l2$ $t$, where $l1$ and $l2$ are the two lights being connected by the road and $t$ is the time (in seconds, $t \leq 500$) to travel the length of the road at full speed — you should add 5 to this value to obtain the travel time when driving the road beginning at a standstill. All roads are two way. At time 0, all lights are just starting their green period and your car is considered to be at a standstill at traffic light $s$. Since it takes 5 seconds to get going, you may assume that $g + y$ is never less than or equal to 5. The last test case is followed by a line containing `0 0 0 0` indicating end-of-input.

### Output

For each test case, output a single line containing the minimum time to travel from the start light to the end light. Output your results in the form `mm:ss` indicating the number of minutes and seconds the trip takes. If the number of seconds is less than 10 then preface it with a 0 (i.e., output `4:05`, not `4:5`). Likewise, if the number of minutes is less than 10, print just one digit (as in `4:05`).

### Sample Input

```
3 3 0 2
3 4 5
3 3 3
2 4 4
0 1 1
1 2 2
0 2 12
3 3 0 2
3 4 5
3 4 3
2 4 4
0 1 1
1 2 2
0 2 12
0 0 0 0
```

### Sample Output

```
0:16
0:08
```