Onsite Practice Contest

ACM ICPC, CSE, BUET

Problems: A – F (12 pages including Cover)

02-Sep-14

Problem A
Mean

You are given a int[] elements . If the arithmetic mean of a non-empty subset of elements is between L and H , inclusive, the subset is considered "good". A subset of a int[] is obtained by removing 0 or more elements from the int[]. Return the number of "good" subsets.

Input

The input contains multiple test cases, until the end of file. Each test case is given in the following order:

- A single line containing L and H
- A single line containing the length of elements
- The elements of the array elements a single line separated by a single space

The elements array will contain between 1 and 36 elements, inclusive. Each element of the elements array will be between -25000000 and 25000000, inclusive. Each element will be distinct. L and H will each be between -25000000 and 25000000, inclusive. L will not be greater than H.

Output

For each test case, output the result in a single line.

Sample Input	Sample Output
2 6 3 10 1 3 -1 0 1 0 100 100 1 0 3 7 10 1 2 3 4 5 6 7 8 9 10	4 1 0 949

Problem B
Root

An integer polynomial of degree n is a function of the form $a_0 + a_1x^1 + a_2x^2 + ... + a_nx^n$, where each a_i is a constant integer and x is a variable. An integer root of an integer polynomial is an integer value of x for which the expression equals zero.

You will be given the coefficients of an integer polynomial, and must return all the integer roots in increasing order. Roots must appear only once in the output (see example 1 for clarification).

Since the degree may be quite large, the coefficients are presented indirectly. Use the following pseudo-code to generate the coefficients a[0] to a[n]:

```
lX = length(X)
lY = length(Y)
for i = 0, 1, ..., n:
        p = i mod lX
        q = (i + Y[i mod lY]) mod lX
        a[i] = X[p]
        X[p] = X[q]
        X[q] = a[i]
```

The array indices are all 0-based and a mod b is the remainder when a is divided by b.

Input

The input consists of several test cases until the end of file. Each test case is given in the following order

- A single line containing n
- A single line containing the length of X
- The elements of the array X in a single line, separated by a single space
- A single line containing the length of Y
- The elements of the array Y in a single line separated by a single space

n will be between 0 and 10,000, inclusive. X will contain between 1 and 50 elements, inclusive. Y will contain between 1 and 50 elements, inclusive. Each element of X will be between -109 and 109, inclusive. Each element of Y will be between 0 and 50, inclusive. At least one element of a will be non-zero.

Output

For each test case, output contains the elements of the resulted array in a single line separated by a single space. For an empty array output the blank line.

Note that, the intended solution is independent of the method of generation, and will work for any integer polynomial.

Sample Input	Sample Output
2 3 -4 2 2 1 0 3 3 1 2 0 4 2 0 0 0 2 3 1 4 4 1 0 3 4 -15 -10 2 1 1 0	-2 1 -1 3

Problem C
Snakes

The Cartesian plane is covered with snakes. You will be given a rectangular portion of the plane, divided into a grid of squares. Each square will either contain a segment of a snake or a barrier.

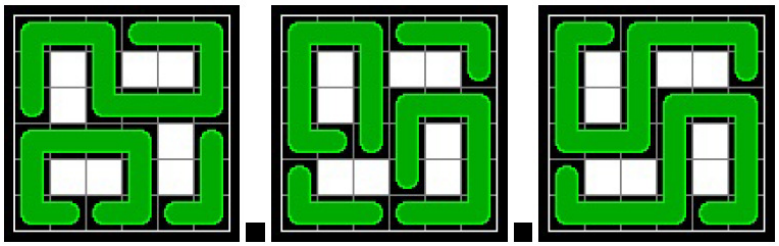
Each snake occupies a chain of adjacent squares, connected horizontally and/or vertically. Snakes are at least 2 segments long (one segment per square) and cannot overlap each other or themselves. Each snake must meet at least one of the following two conditions (see the figures below for examples):

1. Both endpoints must be in squares on the edge of the rectangle.
2. Both endpoints must be horizontally or vertically adjacent to each other, and the snake is at least 4 segments long (so the snake forms a loop).

The portion of the plane will be given as a String grid, where each character represents one square. A '#' represents a barrier and a '.' (period) represents a square that contains a segment of a snake. Fill the grid with snakes so that all of the non-barrier squares are filled. Do this in such a way that minimizes n, the number of snakes whose endpoints are not adjacent (and therefore, must be on the edge of the grid). Return n, or return -1 if there is no way to fill every non-barrier square.

```
{ "......",
".#.##.",
".#....",
"....#.",
".##.#.",
"......" }
```

can be filled with snakes in several ways and a few are shown in the figures below:



Input

The input contains several cases, until the end of file. Input for each test case is given in the following order:

- A single line containing the length of grid
- The elements of the array grid each in a single line

The grid will contain between 1 and 12 elements, inclusive. Each element of grid will contain the same number of characters, between 1 and 12, inclusive. Each element of grid will contain only the characters '#' and '.' (period).

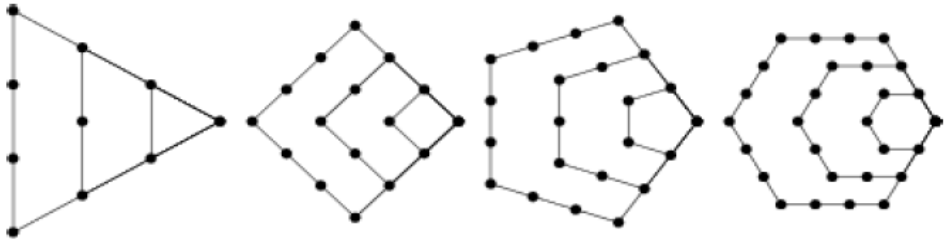
Output

For each test case, output contains the result in a single line.

Sample Input	Sample Output
6 ..... .#.##. .#.... ....#. .##.#. ..... 5 ###.### ###.### ..... ###.### ###.### 7 #.....## ...#.... ....#.#.#. .....#. ...#..#. #..... 8 ########## #.....#... #.####.#... #.#..#.#### #.#..#.#### #.####.#..# #.....#..# ##########	2 -1 1 0

Problem D
Poly-polygonal Numbers

A polygonal number is a number which can be represented by a regular geometrical arrangement of equally spaced points, where the arrangement forms a regular polygon. Some examples are shown in the figure below:



The first figure shows the first 4 *triangular* numbers 1, 3, 6, 10. The next three show the first four *square*, *pentagonal* and *hexagonal* numbers, respectively. In general, *k-gonal* numbers are those whose points define a regular *k-gon* (hence triangular numbers are 3-gonal, square numbers are 4-gonal, etc.). We will define *k* as an *index* of the polygonal number. For this problem, you are to find numbers which are *k*-gonal for two or more values of *k*. We will call these numbers *poly-polygonal*.

Input

Input will consist of multiple problem instances. Each instance will consist of 3 lines. The first line will be a non-negative integer $n <= 50$, indicating the number of types of polygonal numbers of interest in this problem. Note that this line may be longer than 80 characters. The next line will contain *n* integers indicating the indices of these polygonal numbers (all distinct and in increasing order). For example, if the first line contained the value 3, and the next line contained the values 3 6 10, then that problem instance would be interested in 3-gonal, 6-gonal and 10-gonal numbers. Each index *k* will lie in the range $3 <= k <= 1000$. The last line of the problem instance will consist of a single positive integer $s <= 10000$, which serves as a starting point for the search for poly-polygonal numbers. A value of $n = 0$ terminates the input.

Output

For each problem instance, you should output the next 5 poly-polygonal numbers which are greater than or equal to *s*. Each number should be on a single line and conform to the following format:

num:k1 k2 k3 ...

where num is the poly-polygonal number, and k1, k2, k3 ... are the indices (in increasing order) of the poly-polygonal number equal to num. A single space should separate each index, and you should separate each problem instance with a single blank line. The judges input will be such that the maximum value for any poly-polygonal number will fit in a long variable.

Sample Input	Sample Output
10 6 7 8 9 10 11 12 13 14 15 1000 5 3 4 13 36 124 1 0	1216:9 12 1540:6 10 1701:10 13 2300:11 14 3025:12 15 1:3 4 13 36 124 36:3 4 13 36 105:3 36 171:3 13 1225:3 4 124

Problem E
Unjumpers

Unjumpers is a puzzle played on a board consisting of 100 squares in a straight line. Pawns are placed in a certain pattern on the board, and your goal is to see which other patterns can be created starting from that position. There are 3 legal moves in Unjumpers:

1. Jump: A pawn jumps over an adjacent pawn and lands in the square immediately beyond the jumped pawn (in the same direction). The jumped pawn is removed from the board. To perform this move, there must be an adjacent pawn to jump, and the square in which the pawn lands must be unoccupied.
2. Unjump: A pawn jumps over an adjacent empty space and lands in the square immediately beyond that space (in the same direction). A new pawn appears in the square that was jumped (between the starting and ending squares). To perform this move, both the middle and ending squares must be unoccupied.
3. Superjump: A pawn moves 3 squares in one direction. To do this move, the target square must be empty. The two jumped squares may or may not have pawns - and they are not affected by the move.

Only one pawn can move at a time, and pawns may never move off of the board.

You are given a String **start** containing the initial layout of the board. Each character of the string describes one square, with the first character describing the leftmost square. In the string, '.' represents an empty space while '*' represents a pawn. You are also given a String[] targets, each element of which is a target layout formatted in the same way. The board is always 100 squares wide. The Strings given will specify up to 50 of the first (leftmost) squares of the layout. You must assume that the remaining squares are all empty, both when considering the start position and when considering the various target positions.

For each target layout, evaluate whether that layout is reachable using any number of legal moves starting at the initial layout each time. Return the number of target layouts that can be reached.

Input

Input consists of multiple test cases, until the end of file. Each test case is given in the following order:

- A single line containing **start**
- A single line containing the length of **targets**
- The elements of the array **targets** each in a single line

Constraints

- start will contain between 1 and 50 characters, inclusive.
- start will contain only '*' and '.' characters.
- targets will contain between 1 and 50 elements, inclusive.

- Each element of targets will contain between 1 and 50 characters, inclusive.
- Each element of targets will contain only '*' and '.' characters.

Output

For each test case, output the result in a single line.

Sample Input	Sample Output
. 3 ..* *. .*.* ..*** 3 ..****..* ..***.... ..**** *..* 6 *..*..... *.....*... ...*.....* ...*..*... *........* *...***..* ...*** 4 ***... ..**** **....** .*.*.*	3 2 6 3

Problem F
Tiling the Plane

A polygon is said to "tile the plane" if a collection of identical copies of the polygon can be assembled to fill an unbounded two-dimensional plane without any gaps or overlap. For example, Figure 1 shows an L-shaped polygon, and Figure 2 shows how a portion of the plane can be tiled with copies of the polygon. You must write a program to determine whether a given polygon can tile the plane.

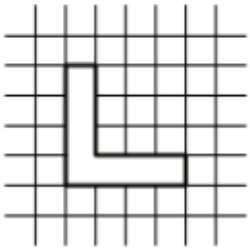


Figure 1: A test polygon shown against a grid of unit squares

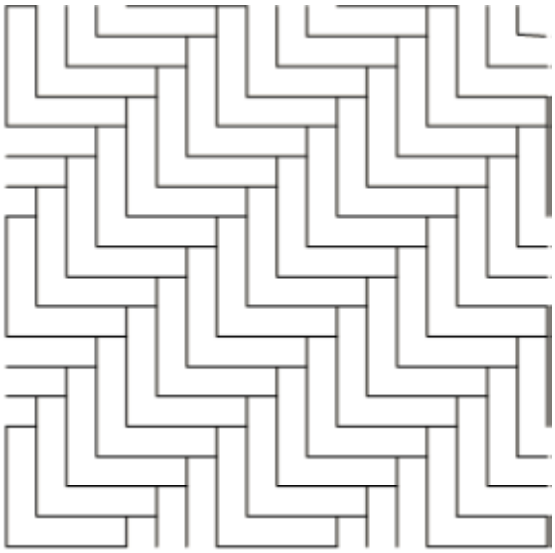


Figure 2: A portion of the plane tiled with the test polygon

Each test case consists of a closed polygon in which every vertex is at a right angle and the length of every side is an integer multiple of a unit length. You may make as many copies of the polygon as you like, and you may move them over the plane, but you may not rotate or reflect any polygon.

You might find the following information useful: It is known that there are only two fundamentally different tilings of the plane, the regular tiling by squares (chessboard tiling) and the tiling by regular hexagons (honeycomb tiling). A polygon can therefore tile the plane if and only if it satisfies one of the following two conditions:

1. There are points A, B, C, D in order on the polygon boundary (the points are not necessarily vertices of the polygon) such that the polygon boundaries from A to B and from D to C are congruent and the boundaries from B to C and from A to D are congruent. This leads to a tiling equivalent to the square tiling.
2. There are points A, B, C, D, E, F in order on the polygon boundary, such that the boundary pairs AB and ED, BC and FE, CD and AF are congruent. This leads to a tiling equivalent to the hexagon tiling.

Input

The input contains the descriptions of several polygons, each description consisting of one input line. Each description begins with an integer n ($4 \le n \le 50$) that represents the number of sides of the polygon. This number is followed by descriptions of n line segments which (taken in order) form a counterclockwise traversal of the perimeter of the polygon. Each line segment description consists of a letter followed by an integer. The letter is "N", "E", "S", or "W", representing the direction of the line segment as North, East, South, or West, respectively. The integer represents the length of the line segment as a multiple of the unit length. The described polygon will not touch or intersect itself.

The input is terminated by a line consisting of the integer zero.

Output

For each polygon in the input, print one output line. Print the number of the polygon in the input, followed by the word "Possible" if it is possible to tile the plane with the test polygon, or "Impossible" otherwise. Follow the format of the sample output.

Sample Input	Sample Output
6 N 3 W 1 S 4 E 4 N 1 W 3 8 E 5 N 1 W 3 N 3 E 2 N 1 W 4 S 5 0	Polygon 1: Possible Polygon 2: Impossible