Day 1 Problem 1

You Can't get There from Here

Source file: there.c *or* there.cpp *or* there.pas
Input file: there.in
Output file: there.out

In a primitive video game, a spot bounces around within a rectangular grid. The southwest corner of the grid has coordinates (0,0) and the northeast corner has coordinates (r,c) where 0 < r <= 10 and 0 < c <= 10. The southeast corner has coordinates (0,c) and the northwest corner has coordinates (r,0). The spot always travels on the diagonal; that is, in one of the directions NE, NW, SE, SW. The outer edges of the grid serve as mirrors: after visiting a position on the edge of the grid the spot "bounces" off according to the normal rules of reflection (Snell's Law). For example, if the spot were travelling NE and hit the east edge of the grid, it would change direction to NW. If the spot were to hit the corner of the grid it would change to the opposite direction.

Given a grid size, two points A and B lying on the grid, and an initial direction, you are to determine if the spot moves from A to B and, if so, how far the spot moves (in terms of number of grid positions) before reaching B the first time.

Input Specification

The input consists of an integer *n*, followed by *n* data sets. Each data set begins with a line containing *r* and *c*, followed by two lines containing the coordinates of points A and B respectively, followed by one line containing NE, NW, SE, or SW - the initial direction of travel.

Output Specification

For each case, print a sentence as shown below indicating whether or not B can be reached, and, if it can, how far the spot moves before reaching B.

Sample Input

```
2
3 4
0 0
0 4
NE
4 2
3 1
3 2
NW
```

Output for Sample Input

```
B can be reached from A after 12 move(s).
B cannot be reached from A.
```

Day 1 Problem 2

Common Words

Source file: words.c *or* words.cpp *or* words.pas
Input file: words.in
Output file: words.out

Given a sequence of *m* words from a newspaper article and an integer *k*, find the *k*th most common word.

Input Specification

Input will consist of an integer *n* followed by *n* data sets. Each data set begins with a line containing *m* and *k*, followed by *m* lines, each containing a word of up to 20 lower case letters. There will be no more than 1000 words per data set.

Output Specification

For each input data set, determine the *k*th most common word(s). To be precise, a word *w* is the *k*th most common if exactly *k-1* distinct words occur more frequently than *w* in the data set. Note that *w* might be multiply defined (i.e. there is a tie for the *k*th most common word) or *w* might not exist (i.e. there is no *k*th most common word). For each data set, print a title line indicating *k* using normal ordinal notation (1st, 2nd, 3rd, 4th, 5th, ...) followed by a number of lines giving all the possible values for the *k*th most common word. A blank line should follow the last word for each data set.

Sample Input

```
3
7 2
the
brown
the
fox
red
the
red
1 3
the
2 1
the
wash
```

Output for Sample Input

```
2nd most common word(s):
red

3rd most common word(s):

1st most common word(s):
the
wash
```

Day 1 Problem 3

Mandelzuma's Revenge

Source file: fract.c *or* fract.cpp *or* fract.pas
Input file: fractal.in
Output file: fractal.out

You may recall that in stage 1 of the CCC you were asked to calculate a region of a fractal obtained by starting with a square, dividing it into 9 sub-squares, then clearing the middle, like so:

```
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * *       * * *
* * *       * * *
* * *       * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
```

This operation was then repeated for each of the 8 remaining sub-squares, and so on.

If we choose a different operation we get a different fractal; for example, we may clear the upper-left and lower-right corners instead of the middle. Or we may divide the square into a different number of sub-squares.

For this problem we will specify the fractal operation as an n by n grid of characters, where each character performs a particular substitution on the corresponding sub-square. The character . (period) indicates that the corresponding sub-square becomes completely empty. The character ! (exclamation mark) indicates that the corresponding sub-square becomes full and is not subject to further fractal operations. The character ? (question mark) indicates that the corresponding sub-square remains full and is subject to further fractal operations.

The fractal above would be represented by the 3 by 3 grid:

```
???
?.?
???
```

Four iterations of the fractal denoted by the 2 by 2 grid

```
.!
?.
```

would be:

```
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
```

Your mission, should you choose to accept it (like you have any choice!) is to print out a region of an arbitrary fractal specified by an *n* by *n* grid. Note that these fractals will get *way* too big to fit completely into memory, so some cleverness will be required to compute only the part to be printed. (Part marks can still be earned for programs that work only on small examples.)

Input Specification

The first line contains a positive integer *n* not exceeding 100. The next *n* lines contain *n* characters each, one of '.', '!', '?', defining the fractal operation to be iterated. The remainder of the input consists of queries for regions of the fractal. Each query consists of two lines: the first line gives *k*, the number of iterations applied to the object (assume that iteration 0 is a completely filled square of size *n^k* by *n^k*); the second line gives *b, t, l, r* specifying the bottom and top row and left and right column of the region to be printed. *k* will not exceed 100, and *b, t, l, r* will not exceed one million. The width and height of the region to be printed will not exceed one hundred. Note that rows are numbered from bottom to top (starting from 1) and columns from left to right (also starting from 1). Input is terminated by a line containing -1.

Output specification

Print the region of the fractal, one line per row. Print the top row first and the bottom row last. Output a '*' for a filled-in portion and a space for a blank section. To make the output appear square, leave a single horizontal space between elements. Leave a blank line in the output after each region.

Sample Input

```
2
. !
?.
3
2 8 1 7
-1
```

Output for Sample Input

```
        * * *
        * * *
        * * *
        * * *
    * *
    * *
  *
```