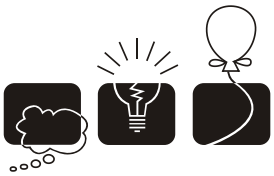


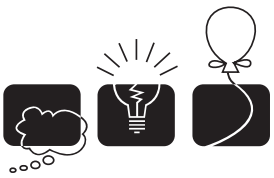
The 34th Annual ACM International Collegiate Programming Contest

Sponsored by IBM

Arab and North Africa Twelfth Regional Contest

Arab Academy for Science and Technology
Alexandria, Egypt
November 2009

The problem set is made of 15 numbered pages



[A] Seinfeld

Program:	`braces.(c|cpp|java)`
Input:	`braces.in`
Balloon Color:	Red

Description

I'm out of stories. For years I've been writing stories, some rather silly, just to make simple problems look difficult and complex problems look easy. But, alas, not for this one.

You're given a non empty string made in its entirety from opening and closing braces. Your task is to find the minimum number of *"operations"* needed to make the string *stable.* The definition for being stable is as follows:

1. An empty string is stable.
2. If S is stable, then $\{S\}$ is also stable.
3. If S and T are both stable, then ST (the concatenation of the two) is also stable.

All of these strings are stable: {}, {}{}, and {{}{}}; But none of these: }{, {{}{, nor {}{.

The only operation allowed on the string is to replace an opening brace with a closing brace, or visa-versa.

Input Format

Your program will be tested on one or more data sets. Each data set is described on a single line. The line is a non-empty string of opening and closing braces and nothing else. No string has more than 2000 braces. All sequences are of even length.

The last line of the input is made of one or more `'-'` (minus signs.)

Output Format

For each test case, print the following line:

`k.␣N`

Where `k` is the test case number (starting at one,) and `N` is the minimum number of operations needed to convert the given string into a balanced one.

Sample Input/Output

braces.in

```
}{
{}{}{}
{{{}
---
```

OUTPUT

```
1. 2
2. 0
3. 1
```

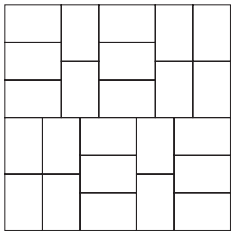
[B] Tiles of Tetris, NOT!

Program:	`tiles.(c|cpp|java)`
Input:	`tiles.in`
Balloon Color:	Orange

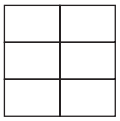
Description

You've really messed up this time. *"Go buy some square tiles"* your supervisor told you. But as usual, you were either busy reading that message, answering that e-email, or updating your wall status on facebook. *"Go buy some tiles"* was all that you could remember. Your supervisor is now complaining that the tiles you bought were not squares and you'll be fired if you don't *fix it!*

"Fix it? How am I supposed to fix it?" you say to yourself. *"I'm a programmer, The only tiles I know are those of the game Tetris!"* There is no way you can afford to be fired either. This is your third job in less than a month. *"I have to fix it!"* you keep repeating.



You call the store you bought the tiles from, but they won't take them back or even exchange. *"We cannot cancel orders once the merchandise has left the store",* they tell you. You take a look at the tiles, they're all rectangle-shaped and all of the same size. You'd be losing a lot if you cut them, not that you'd be able to cut them into perfect squares in the first place. *"Wait a minute!"* you say to yourself. You grab a bunch of tiles, put them in some arrangement and *Voila, that's a square.* But that won't convince your boss. Tiles are supposed to be small, and what you came-up with is rather big. You take another look at your arrangement, and it hits you again. *"I'm on a roll today!"* you say proudly to yourself. You just figured out how to find the smallest number of tiles needed to form the smallest possible square.



You rush and bring your supervisor to show him your discovery. He's not that much impressed. You're not making sense anyway (given all the blood rushing in your head trying to explain your algorithm.) You know that the calculation is definitely much simpler than what you're saying, but you just can't seem to think clearly. Finally, your supervisor shouts: *"First of all, the tiles have to be laid-down in the same orientation. Second, I'm running a construction site here, not a software shop! How do you expect the workers to figure out that number! You either write me a program to do the calculation you're describing, or you go collect your netbook, your cell, your ipod, and your blackberry, and you get out of here this minute!"*

Input Format

Your program will be tested on one or more data sets. Each test set is described on a single line made of two positive numbers: $(0 < W, H < 1,000,000)$ which are the width and height of each tile. The last line is made of two zeros.

Output Format

For each test case, write the answer on a separate line.

Sample Input/Output

tiles.in

```
2 3
1 2
0 0
```

OUTPUT

```
6
2
```

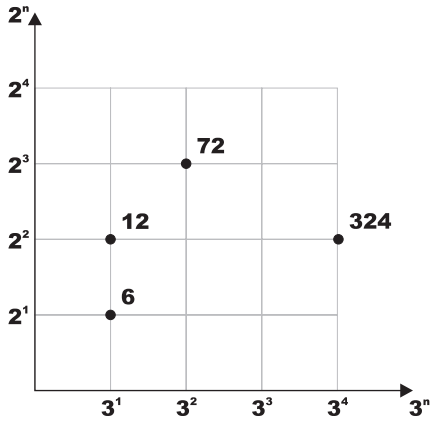
[C] Not So Flat After All

Program:	`space.(c|cpp|java)`
Input:	`space.in`
Balloon Color:	`Green`

Description

Any positive integer v can be written as $p_1^{a_1} * p_2^{a_2} * \ldots * p_n^{a_n}$ where p_i is a prime number and $a_i \geq 0$. For example: $24 = 2^3 * 3^1$.

Pick any two prime numbers p_1 and p_2 where $p_1 \neq p_2$. Imagine a two dimensional plane where the powers of p_1 are plotted on the x-axis and the powers of p_2 on the y-axis. Now any number that can be written as $p_1^{a_1} * p_2^{a_2}$ can be plotted on this plane at location $(x, y) = (a_1, a_2)$. The figure on the right shows few examples where $p_1 = 3$ and $p_2 = 2$.

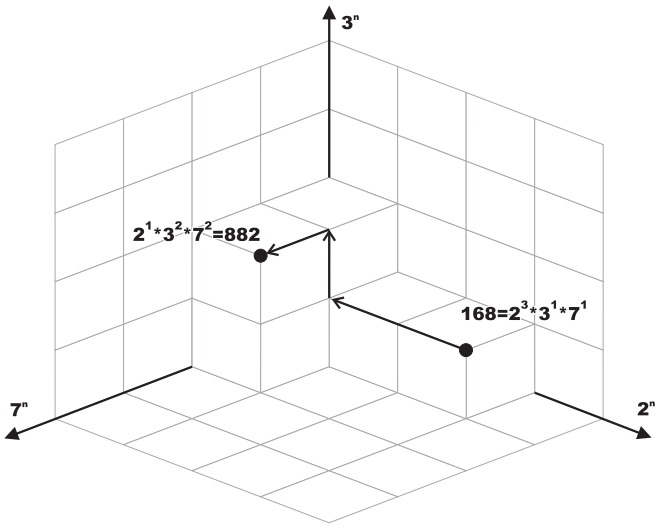


This idea can be extended for any N-Dimensional space where each of the N axes is assigned a unique prime number. Each N-Dimensional space has a unique set of primes. We call such set the *Space Identification Set* or S for short. $|S|$ (the ordinal of S) is N.

Any number that can be expressed as a multiplication of $p_i \in S$ (each raised to a power $(a_i \geq 0)$ can be plotted in this $|S|$-Dimensional space. The figure at the bottom illustrates this idea for $N = 3$ and $S = \{2, 3, 7\}$. Needless to say, any number that can be plotted on space `A` can also be plotted on space `B` as long as $S_A \subset S_B$.

We define the distance between any two points in a given N-Dimensional space to be the sum of units traveled to get from one point to the other while following the grid lines (i.e. movement is always parallel to one of the axes.) For example, in the figure below, the distance between 168 and 882 is 4.

Given two positive integers, write a program that determines the minimum ordinal of a space where both numbers can be plotted in. The program also determines the distance between these two integers in that space.



Input Format

Your program will be tested on one or more test cases. Each test case is specified on a line with two positive integers ($0 < A, B < 1,000,000$) where $A * B > 1$.

The last line is made of two zeros.

Output Format

For each test case, print the following line:

`k.␣X:D`

Where `k` is the test case number (starting at one,) `X` is the minimum ordinal needed in a space that both A and B can be plotted in. `D` is the distance between these two points.

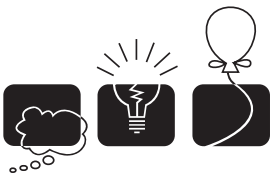
Sample Input/Output

space.in

```
168 882
770 792
0 0
```

OUTPUT

```
1. 3:4
2. 5:6
```



[D] Probability One

Program:	`guess.(c|cpp|java)`
Input:	`guess.in`
Balloon Color:	`Blue`

Description

Number guessing is a popular game between elementary-school kids. Teachers encourage pupils to play the game as it enhances their arithmetic skills, logical thinking, and following-up simple procedures. We think that, most probably, you too will master in few minutes. Here's one example of how you too can play this game: Ask a friend to think of a number, let's call it n_0. Then:

1. Ask your friend to compute $n_1 = 3 * n_0$ and to tell you if n_1 is even or odd.
2. If n_1 is even, ask your friend to compute $n_2 = n_1/2$. If, otherwise, n_1 was odd then let your friend compute $n_2 = (n_1 + 1)/2$.
3. Now ask your friend to calculate $n_3 = 3 * n_2$.
4. Ask your friend to tell tell you the result of $n_4 = n_3/9$. (n_4 is the quotient of the division operation. In computer lingo, '/' is the integer-division operator.)
5. Now you can simply reveal the original number by calculating $n_0 = 2 * n_4$ if n_1 was even, or $n_0 = 2 * n_4 + 1$ otherwise.

Here's an example that you can follow: If $n_0 = 37$, then $n_1 = 111$ which is odd. Now we can calculate $n_2 = 56$, $n_3 = 168$, and $n_4 = 18$, which is what your friend will tell you. Doing the calculation $2 \times \ n_4 + 1 = 37$ reveals n_0.

Input Format

Your program will be tested on one or more test cases. Each test case is made of a single positive number ($0 < n_0 < 1,000,000$).

The last line of the input file has a single zero (which is not part of the test cases.)

Output Format

For each test case, print the following line:

`k.␣B␣Q`

Where `k` is the test case number (starting at one,) `B` is either `'even'` or `'odd'` (without the quotes) depending on your friend's answer in step 1. `Q` is your friend's answer to step 4.

Sample Input/Output

guess.in

```
37
38
0
```

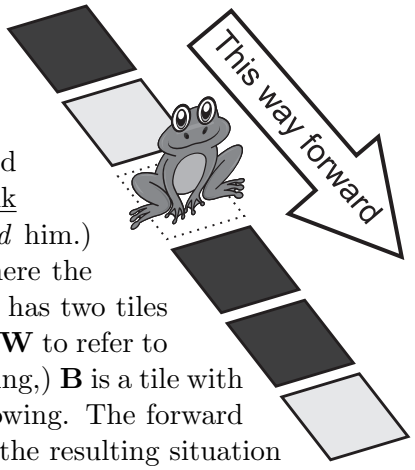
OUTPUT

```
1. odd 18
2. even 19
```

[E] Hop — Don't Walk!

Program:	`hop.(c|cpp|java)`
Input:	`hop.in`
Balloon Color:	Purple

Description



Kermit The Frog is a classic video game with a simple control and objective but requires a good deal of thinking. You control an animated frog that can walk and hop, in both forward and backward directions. The frog stands in a *space* between an otherwise a contiguous line of *tiles.* Each tile is painted black on one side, and white on the other. The frog can walk (forward, or backward) over an adjacent tile (*in front* or *behind* him.) When the frog walks over a tile, the tile *slides* to the space where the frog was standing. For example, in the adjacent figure, the frog has two tiles behind him, and three in front. We'll use the notation **BWFBBW** to refer to this situation where **F** refers to the space (where the frog is standing,) **B** is a tile with its black face showing, while **W** is a tile with its white face showing. The forward direction is from left to right. If the frog were to walk forward, the resulting situation is **BWBFBW**. Similar behavior when the frog walks backward, the tile behind the frog slides to where the frog was standing. The frog can also hop over the tiles. The frog can hop over an adjacent tile landing on the tile next to it. For example, if the frog was to hop backward, it would land on the first (left-most) tile, and the tile would jump to the space where the frog was standing. In addition, *the tile would flip sides.* For example, hopping backward in the figure would result in the situation: **FWWBBW**. We challenge you to write a program to determine the minimum number of moves (walks or hops) to transform one tile configuration into another.

Input Format

Your program will be tested on one or more test cases. Each test case is specified on a single line that specifies string S representing the initial tile arrangement. S is a non-empty string and no longer than 100 characters and is made of the letters `'B'`, `'W'`, and exactly one `'F'`. The last line of the input file has one or more `'-'` (minus) characters.

Output Format

For each test case, print the following line:

`k.␣M`

Where `k` is the test case number (starting at one,) and `M` is the minimum number of moves needed to transform the given arrangement to an arrangement that has no white tile(s) between any of its black tiles . The frog can be anywhere. `M` is -1 if the problem cannot be solved in less than 10 moves.

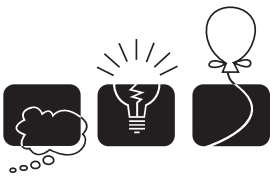
Sample Input/Output

hop.in

```
WWBBFBW
WWFBWBW
FWBBWBW
---
```

OUTPUT

```
1. 0
2. 1
3. 2
```



[F] Air Strike

Program:	`strike.(c|cpp|java)`
Input:	`strike.in`
Balloon Color:	Brown

Description

General Gee is the commander of a military base. He has just received alarming news from one of his spies: the enemy's preparing an air missile strike. The base contains two magnetic towers. When activated and given sufficient power, each of the magnetic towers creates a powerful horizontal magnetic disk. If any missile passes through this disk it deflects away from the base. Although those towers seem to be an excellent air defense method, there is a problem: The area of the disk generated by a tower is proportional to the amount of energy it receives. The base has enough power plants to generate a certain amount of energy, which has to be divided among those two towers. That means that the total area of the two disks generated from the towers should not exceed the total energy generated by the power plants. Fortunately, the spy was able to know the exact target co-ordinates of the incoming missiles and he reported them to General Gee. The General needs your help in distributing the energy on the two magnetic towers to minimize the number of missiles that will not get deflected by the magnetic towers and therefore will hit the base. You may assume the following:

- The towers have different heights and therefore there are no problems associated with the magnetic disks interfering with each other.
- A missile will deflect if it passes through the magnetic disk of a tower or even if it just touches its boundary.
- A missile hitting a tower (landing exactly on its location) will deflect, even if the tower is not given any energy.
- All incoming missiles will go down simultaneously at the exact instant; therefore, there will not be any time available to redistribute the energy amongst the two towers during the strike.

Input Format

Input consists of several test cases. Each test case is specified on $N+2$ lines. The first line contains an integer ($1 \leq N \leq 1,000$) representing the number of missiles. The second line contains 5 real numbers X_1, Y_1, X_2, Y_2 and T: (X_1, Y_1) is the coordinates of the first tower, (X_2, Y_2) is the coordinates of the second tower and $(0 \leq T)$ is the total amount of energy generated from the power plants (the total area of the two magnetic disks). Each line of the remaining N lines contains two real numbers representing the landing coordinates of a missile.

The absolute value of all the given real numbers is less than or equal to 100 and may include a decimal point followed by up to 3 digits. Any two consecutive numbers on the same line are separated by one or more white-space characters. Zero or more blank lines may appear between test cases.

The last line of the input file is made of a single zero.

Output Format

For each test case, print the following line:

`k.␣M`

Where `k` is the test case number (starting at one,) and `M` is the minimum number of missiles that will NOT be deflected in the best distribution of energy among the two towers. Use $\pi = 3.141$.

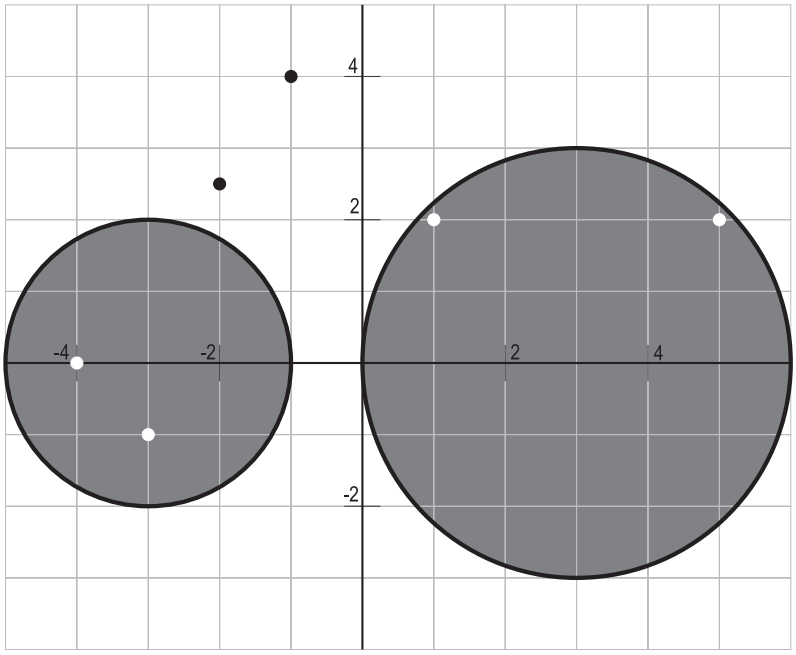
Sample Input/Output

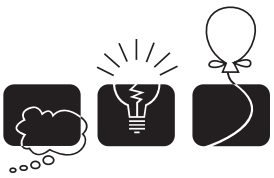
strike.in

```
6
-3 0  3 0  40.833
-1 4
-2 2.5
1 2
5 2
-4 0
-3 -1

2
0 0  1 1  0
0 0
1 1

0
```

OUTPUT

```
1. 2
2. 0
```





[G] Stock Chase

Program:	`stock.(c|cpp|java)`
Input:	`stock.in`
Balloon Color:	Pink

Description

I have to admit, the solution I proposed last year for solving the bank cash crisis didn't solve the whole economic crisis. As it turns out, companies don't have that much cash in the first place. They have assets which are primarily shares in other companies. It is common, and acceptable, for one company to own shares in another. What complicates the issue is for two companies to own shares *in each other at the same time*. If you think of it for a moment, this means that each company now (indirectly) controls its own shares.

New market regulation is being implemented: No company can control shares in itself, whether directly or indirectly. The Stock Market Authority is looking for a computerized solution that will help it detect any buying activity that will result in a company controlling its own shares. It is obvious why they need a program to do so, just imagine the situation where company A buying shares in B, B buying in C, and then C buying in A. While the first two purchases are acceptable. The third purchase should be rejected since it will lead to the three companies controlling shares in themselves. The program will be given all purchasing transactions in chronological order. The program should reject any transaction that could lead to one company controlling its own shares. All other transactions are accepted.

Input Format

Your program will be tested on one or more test cases. Each test case is specified on $T + 1$ lines. The first line specifies two positive numbers: $(0 < N \leq 234)$ is the number of companies and $(0 < T \leq 100,000)$ is the number of transactions. T lines follow, each describing a buying transaction. Each transaction is specified using two numbers A and B where $(0 < A, B \leq N)$ indicating that company A wants to buy shares in company B.

The last line of the input file has two zeros.

Output Format

For each test case, print the following line:

`k.␣R`

Where `k` is the test case number (starting at one,) `R` is the number of transactions that should be rejected.

Sample Input/Output

stock.in

```
3 6
1 2
1 3
3 1
2 1
1 2
2 3
0 0
```

OUTPUT

```
1. 2
```

[H] Land Division

Program:	`land.(c|cpp|java)`
Input:	`land.in`
Balloon Color:	Gold

Description

The king of the *Far, Far Away Kingdom* has passed-away and the kingdom must be split amongst his K sons. The kingdom, which can be drawn on a rectangular map, consists of N cities. To divide the land, they will draw $K-1$ straight segments on the map, all of them parallel to either the vertical or the horizontal axis of the map. This divides the map into exactly K rectangles, all having equal heights (if the dividing lines where vertical), or equal widths (if the dividing lines where horizontal). No segment should pass through any of the cities. Each son will then be assigned one random region out of the K regions and the cities inside that region will be his share.

Of course, they want the division to be as fair as possible: theoretically, in the fairest division, each son should get N/K cities (we'll call this value the baseline), but since the baseline isn't always a whole number, each of the sons wants to be as close as possible to the baseline. We will calculate the unfairness of each son as the absolute difference between the number of cities assigned to him and the baseline. The fairest division is the one that minimizes the average unfairness of all the sons.

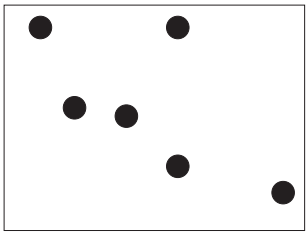


Figure (a)

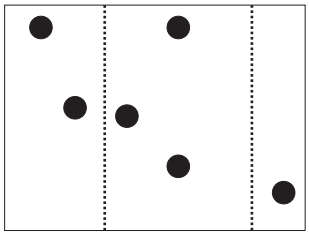


Figure (b)

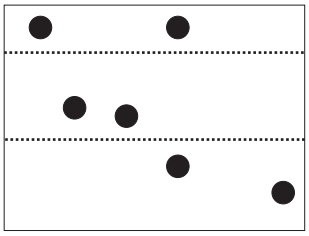


Figure (c)

Consider the example above with 3 sons and 6 cities (so the baseline is $6/3 = 2.0$) Figure (a) is the original map. Figure (b) shows a non-optimal division (the dashed lines are the 2 dividing lines.) In this case, the middle region contains 3 cities (unfairness of $|3-2| = 1$), the left region contains 1 city (unfairness of $|1-2| = 1$), while the right region contains 2 cities (perfectly fair, unfairness of 0), so the average unfairness is $2/3$.

Figure (c) on the right shows the optimal division since all three regions contain the same number of cities for an average unfairness of 0.

Write a program to determine the fairest land division for a given kingdom.

Input Format

Your program will be tested on one or more test cases. Each test case is described on $N+1$ lines. The first line of each test case specifies two positive integers: ($N \leq 100,000$) and ($K \leq 10$) where N is the number of cities and K is number of children. Note that $K \leq N$.

N lines follows, each describing the coordinates of a city by specifying two integers (x, y) where $0 \leq x, y \leq 100,000$. Since coordinates are rounded to the nearest integer, more than one city could have the exact same coordinate on the map. You may assume that the map of the kingdom is any rectangle that contains all of the given points (although such information is not needed by the program.) Note also that while all cities lie on integer coordinates, the dividing lines need not be.

The last line of the input file contains two zeros.

Output Format

For each test case, print the following line:

`k.␣A/B`

Where `k` is the test case number (starting at one,) and `A/B` is the minimum average that could be obtained. `A/B` should be an irreducible fraction. Let `B=1` when the result is a whole number.

Sample Input/Output

land.in

```
6 3
0 4
1 3
2 3
3 1
4 4
5 0
4 3
0 0
0 1
1 1
1 0
0 0
```

OUTPUT

```
1. 0/1
2. 8/9
```

[I] Kind of a Blur

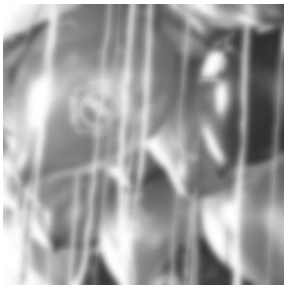
Program:	`blur.(c|cpp|java)`
Input:	`blur.in`
Balloon Color:	`Silver`

Description

Image blurring occurs when the object being captured is out of the camera's focus. The top two figures on the right are an example of an image and its blurred version. Restoring the original image given only the blurred version is one of the most interesting topics in image processing. This process is called deblurring, which will be your task for this problem.

In this problem, all images are in grey-scale (no colours). Images are represented as a 2 dimensional matrix of real numbers, where each cell corresponds to the brightness of the corresponding pixel. Although not mathematically accurate, one way to describe a blurred image is through averaging *all the pixels that are within (less than or equal to) a certain Manhattan distance*[†] *from each pixel (including the pixel itself).* Here's an example of how to calculate the blurring of a 3x3 image with a blurring distance of 1:

Focused Image



Blurred Image

6	5	4	3	4	5	6
5	4	3	2	3	4	5
4	3	2	1	2	3	4
3	2	1	0	1	2	3
4	3	2	1	2	3	4
5	4	3	2	3	4	5
6	5	4	3	4	5	6

Manhattan Distance

$$\text{blur}\left(\begin{bmatrix} 2 & 30 & 17 \\ 25 & 7 & 13 \\ 14 & 0 & 35 \end{bmatrix}\right)$$

$$= \begin{bmatrix} \frac{2+30+25}{3} & \frac{2+30+17+7}{4} & \frac{30+17+13}{3} \\ \frac{2+25+7+14}{4} & \frac{30+25+7+13+0}{5} & \frac{17+7+13+35}{4} \\ \frac{25+14+0}{3} & \frac{7+14+0+35}{4} & \frac{13+0+35}{3} \end{bmatrix}$$

$$= \begin{bmatrix} 19 & 14 & 20 \\ 12 & 15 & 18 \\ 13 & 14 & 16 \end{bmatrix}$$

Given the blurred version of an image, we are interested in reconstructing the original version assuming that the image was blurred as explained above.

[†]The Manhattan Distance (sometimes called the Taxicab distance) between two points is the sum of the (absolute) difference of their coordinates. The grid on the lower right illustrates the Manhattan distances from the grayed cell.

Input Format

Input consists of several test cases. Each case is specified on $H + 1$ lines. The first line specifies three non negative integers specifying the width W, the height H of the blurred image and the blurring distance D respectively where $(1 \leq W, H \leq 10)$ and $(D \leq \min(W/2, H/2))$. The remaining H lines specify the gray-level of each pixel in the blurred image. Each line specifies W non-negative real numbers given up to the 2^{nd} decimal place. The value of all the given real numbers will be less than 100.

Zero or more lines (made entirely of white spaces) may appear between cases. The last line of the input file consists of three zeros.

Output Format

For each test case, print a $W \times H$ matrix of real numbers specifying the deblurred version of the image. Each element in the matrix should be approximated to 2 decimal places and right justified in a field of width 8. Separate the output of each two consecutive test cases by an empty line. Do not print an empty line after the last test case. It is guaranteed that there is exactly one unique solution for every test case.

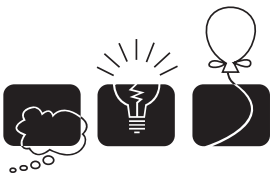
Sample Input/Output

blur.in

```
2 2 1
1 1
1 1

3 3 1
19 14 20
12 15 18
13 14 16

4 4 2
14 15 14 15
14 15 14 15
14 15 14 15
14 15 14 15

0 0 0
```

OUTPUT

```
    1.00    1.00
    1.00    1.00

    2.00   30.00   17.00
   25.00    7.00   13.00
   14.00    0.00   35.00

    1.00   27.00    2.00   28.00
   21.00   12.00   17.00    8.00
   21.00   12.00   17.00    8.00
    1.00   27.00    2.00   28.00
```



[J] National Treasures

Program:	`guards.(c|cpp|java)`
Input:	`guards.in`
Balloon Color:	`Black`

Description

The great hall of the national museum has been robbed few times recently. Everyone is now worried about the security of the treasures on display. To help secure the hall, the museum contracted with a private security company to provide additional guards to stay in the great hall and keep an eye on the ancient artifacts. The museum would like to hire the minimum number of additional guards so that the great hall is secured.

The great hall is represented as a two dimensional grid of $R \times C$ cells. Some cells are already occupied with the museum's guards. All remaining cells are occupied by artifacts of different types (statues, sculptures, ... etc.) which can be replaced by new hired guards. For each artifact, few other cells in the hall are identified as critical points of the artifact depending on the artifact value, type of vault it is kept inside, and few other factors. In other words, if this artifact is going to stay in the hall then all of its critical points must have guards standing on them. A guard standing in a critical position of multiple artifacts can keep an eye on them all. A guard, however, can not stand in a cell which contains an artifact (instead, you may remove the artifact to allow the guard to stay there). Also you can not remove an artifact and leave the space free (you can only replace an artifact with a new hired guard).

Surveying all the artifacts in the great hall you figured out that the critical points of any artifact (marked by a $\otimes$) are always a subset of the 12 neighboring cells as shown in the grid below.

	2		3	
1		9		4
	12	$\otimes$	10	
8		11		5
	7		6	

Accordingly, the type of an artifact can be specified as a non-negative integer where the i-th bit is 1 only if critical point number i from the picture above is a critical point of that artifact. For example an artifact of type 595 (in binary 1001010011) can be pictured as shown in the figure below. Note that bits are numbered from right to left (the right-most bit is bit number 1.) If a critical point of an artifact lies outside the hall grid then it is considered secure.

	2			
1				
		$\otimes$	10	
				5
	7			

You are given the layout of the great hall and are asked to find the minimum number of additional guards to hire such that all remaining artifacts are secured.

Input Format

Your program will be tested on one or more test cases. Each test case is specified using $R+1$ lines. The first line specifies two integers ($1 \leq R, C \leq 50$) which are the dimensions of the museum hall. The next R lines contain C integers separated by one or more spaces. The j-th integer of the i-th row is -1 if cell (i, j) already contains one of the museum's guards, otherwise it contains an integer ($0 \leq T < 2^{12}$) representing the type of the artifact in that cell.

The last line of the input file has two zeros.

Output Format

For each test case, print the following line:

`k.␣G`

Where `k` is the test case number (starting at one,) and `G` is the minimum number of additional guards to hire such that all remaining artifacts are secured.

Sample Input/Output

guards.in

```
1 3
512 -1 2048
2 3
512 2560 2048
512 2560 2048
0 0
```

OUTPUT

```
1. 0
2. 2
```

The picture on the right shows the solution of the second test case where the two artifacts in the middle are replaced by guards.