2011 Mid-Atlantic Regional Programming Contest

Welcome to the 2011 ICPC Mid-Atlantic Regional. Before you start the contest, please be take the time to review the following:

The Contest

1. There are eight (8) problems in the packet, using letters A–H. These problems are NOT sorted by difficulty. As a team's solution is judged correct, the team will be awarded a balloon. The balloon colors are as follows:

Problem	Problem Name	Balloon Color
A	A Most Ingenious Pair a' Twins	Yellow
B	Raggedy, Raggedy	Green
C	AAAA	Silver
D	Thunk and Plunk	Black
E	Losers Are Winners	Orange
F	Line of Sight	Purple
G	Stained Glass	Pink
H	Road Rally	Red

2. Solutions for problems submitted for judging are called runs. Each run will be judged.

   The judges will respond to your submission with one of the following responses. In the event that more than one response is applicable, the judges may respond with any of the applicable responses.

Response	Explanation
Correct	Your submission has been judged correct.
Wrong Answer	Your submission generated output that is not correct or is incomplete.
Output Format Error	Your submission's output is not in the correct format or is misspelled.
Excessive Output	Your submission generated output in addition to or instead of what is required.
Compilation Error	Your submission failed to compile.
Run-Time Error	Your submission experienced a run-time error.
Time-Limit Exceeded	Your submission did not solve the judges' test data within 30 seconds.

3. A team's score is based on the number of problems they solve and penalty points, which reflect the amount of time and number of incorrect submissions made before the problem is solved. For each problem solved correctly, penalty points are charged equal to the time at

which the problem was solved plus 20 minutes for each incorrect submission. No penalty points are added for problems that are never solved. Teams are ranked first by the number of problems solved and then by the fewest penalty points.

4. This problem set contains sample input and output for each problem. However, the judges will test your submission against longer and more complex datasets, which will not be revealed until after the contest. Your major challenge is designing other input sets for yourself so that you may fully test your program before submitting your run. Should you receive a judgment stating that your submission was incorrect, you should consider what other datasets you could design to further evaluate your program.

5. In the event that you feel a problem statement is ambiguous or incorrect, you may request a clarification. Read the problem carefully before requesting a clarification. If the judges believe that the problem statement is sufficiently clear, you will receive the response, "The problem statement is sufficient; no clarification is necessary." If you receive this response, you should read the problem description more carefully. If you still feel there is an ambiguity, you will have to be more specific or descriptive of the ambiguity you have found.

   You may not submit clarification requests asking for the correct output for inputs that you provide. Sample inputs *may* be useful in explaining the nature of a perceived ambiguity, e.g., "There is no statement about the desired order of outputs. Given the input: . . . , would both this: . . . and this: . . . be valid outputs?".

   If a clarification is issued during the contest, it will be broadcast to all teams.

6. Runs for each particular problem will be judged in the order they are received. However, it is possible that runs for different problems may be judged out of order. For example, you may submit a run for problem B followed by a run for problem C, but receive the response for C first.

   Do not request clarifications on when a response will be returned. If you have not received a response for a run within 30 minutes of submitting it, **you may have a runner ask the local site judge to determine the cause of the delay. Under no circumstances should you ever submit a clarification request about a submission for which you have not received a judgment.**

   If, due to unforeseen circumstances, judging for one or more problems begins to lag more than 30 minutes behind submissions, a clarification announcement will be issued to all teams. This announcement will include a change to the 30 minute time period that teams are expected to wait before consulting the site judge.

7. The submission of abusive programs or clarification requests to the judges will be considered grounds for immediate disqualification.

8. The submission of code deliberately designed to delay, crash, or otherwise negatively affect the contest itself will be considered grounds for immediate disqualification.

Your Programs

9. All solutions must read from standard input and write to standard output. In C this is scanf/printf, in C++ this is cin/cout, and in Java this is System.in/System.out. The judges will ignore all output sent to standard error (cerr in C++ or System.err in Java). You may wish to use standard error to output debugging information. From your workstation you may test your program with an input file by redirecting input from a file:

   ```
   program < file.in
   ```

10. Unless otherwise specified, all lines of program output

    - should be left justified, with no leading blank spaces prior to the first non-blank character on that line,
    - should end with the appropriate line terminator (`\n`, `endl`, or `println()`), and
    - should not contain any blank characters at the end of the line, between the final specified output and the line terminator.

    You should not print extra lines of output, even if empty, that are not specifically required by the problem statement.

11. If a problem calls for rounding a floating point number to a specific precision, then rounding should be carried out so that trailing digits of 5 or higher are rounded up, trailing digits of 4 or less are rounded down. For example, if rounding to the nearest 0.01 is requested, then 0.0152 would round to 0.02, but 0.0149 would round to 0.01.

12. Unless otherwise specified, all numbers in your output should begin with the minus sign (−) if negative, followed immediately by 1 or more decimal digits. If the number being printed is a floating point number, then the decimal point should appear, followed by the appropriate number of decimal digits. For output of real numbers, the number of digits after the decimal point will be specified in the problem description (as the "precision").

    All floating point numbers printed to a given precision should be rounded to the nearest value.

    In simpler terms, neither scientific notation nor commas will be used for numbers, and you should ensure you use a printing technique that rounds to the appropriate precision.

13. All input sets used by the judges will follow the input format specification found in the problem description. You do not need to test for input that violates the input format specified in the problem.

14. All lines of program input will end with the appropriate line terminator (e.g., a linefeed on Unix/Linux systems, a carriage return-linefeed pair on Windows systems).

15. If a problem specifies that an input is a floating point number, the input will be presented according to the rules stipulated above for output of real numbers, except that decimal points and the following digits may be omitted for numbers with no non-zero decimal portion. Scientific notation will not be used in input sets unless a problem explicitly allows it.

16. Every effort has been made to ensure that the compilers and run-time environments used by the judges are as similar as possible to those that you will use in developing your code. With that said, some differences may exist. It is, in general, your responsibility to write your code in a portable manner compliant with the rules and standards of the programming language. You should not rely upon undocumented and non-standard behaviors.

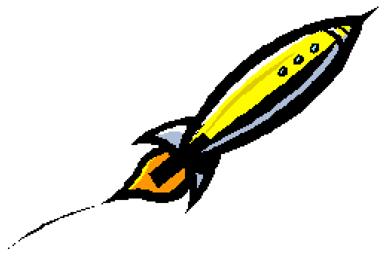
    One place where differences are likely to arise is in the size of the various numeric types. Many problems will specify minimum and maximum values for numeric inputs and outputs. You should write your code with the understanding that, on the *judges'* machines:

    - A C++ `int`, a C++ `long`, and a Java `int` are all 32-bits wide.
    - A C++ `long long` and a Java `long` are 64-bits wide.
    - A `float` in both languages is a 32-bit value capable of holding 6-7 decimal digits, though many library functions will be less accurate.
    - A `double` in both languages is a 64-bit value capable of holding 15-16 decimal digits, though many library functions will be less accurate.

    The data types on your own machines may differ in size from these, but if you follow the guidelines above in choosing the types to hold your numbers, you can be assured that they will suffice to hold those values on the judges' machines.

Good luck, and HAVE FUN!!!

Problem A: A Most Ingenious Pair a' Twins



One of the intriguing predictions of the Special Theory of Relativity is that time actually slows down for moving objects. We don't normally notice this, because the effect is tiny until you get up to a good-sized fraction of the speed of light (299,792,458 meters per second). So it's not something you will notice on your average drive to and from the supermarket.

The implications of this slowdown of time is often illustrated by the so-called *Twins Paradox*. Imagine two twins, both young adults, one of whom remains on Earth while the other boards a rocket ship that travels at nearly the speed of light for 50 years before returning to Earth. The Earth-bound twin, now a senior citizen after 50 years, is astounded to greet the traveler who has aged only a few years.

The relative time experienced by the two twins is given by the formula

$$t_r = \gamma t_e$$

where t_r is the time experienced by the twin on the rocket and t_e is the amount of time experienced by the twin who remains on Earth. The conversion factor γ is given by

$$\gamma = \sqrt{1 - \frac{v^2}{c^2}}$$

where v is the average velocity of the rocket and c is the speed of light (expressed in the same units as v).

For example, if v is $259,620,268$ meters per second, then $\gamma = 0.5$ (approximately) and the Earth-bound twin experiences 2 years for every year experienced by the traveler.

Write a program to determine, for two target values of t_e and t_r, the average velocity that the rocket would need to travel to produce that difference in time experienced by the two twins.

Input

Input will consist of one or more datasets. Each dataset will consist of two floating point values on a single line. These values will denote t_e and t_r, respectively, expressed in years. You are guaranteed that $t_e \geq t_r$ and that both times will be in the range $1.0 \ldots 100.0$, inclusive.

End of input is indicated by a line containing two zeros.

Output

For each dataset, print a single line of output containing the desired velocity, expressed in lightyears per year, printed to a precision of 3 decimals.

Example

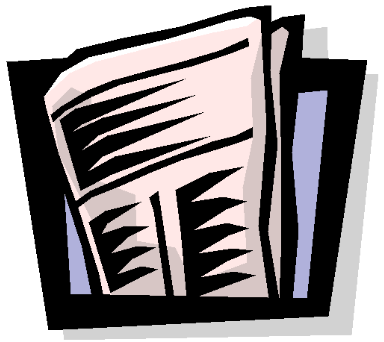
Given the input

```
10.5 5.25
5 1
0 0
```

the output would be

```
0.866
0.980
```

Problem B: Raggedy, Raggedy



Consider the problem of laying out text in lines of a fixed maximum width L (a.k.a., "line filling").

If you do a poor job,
the ends of the lines are unnecessarily
ragged - like
this paragraph. Now,
by convention, we allow the last line of a paragraph to be arbitrarily ragged. We don't mind if that final line contains just a few characters, but we expect the earlier lines to be of approximately uniform length, filling up the column in which we are setting the text.

The straightforward approach of filling each line with as many words as will fit and then moving to the next line does not always yield the most aesthetically pleasing results. For example, the sequence

```
See if we care.
```

could be laid out in L=6 like this:

```
See if
we
care.
```

That layout is, arguably, not as visually pleasing as

```
See
if we
care.
```

Define a "*word*" as any sequence of non-whitespace characters bounded by a line start or end or by a blank. The legal "*whitespace characters*" in this problem are blanks and the line terminator characters.

Given a sequence of N words of width $w_1, w_2, \ldots, w_N$, and a maximum line width L, with the guarantee that for all i, $w_i \geq L$, define $w(i, j)$ as the width of the line containing words i through j, inclusive, plus one blank space between each pair of words.

Then we can define the raggedness of a line containing words i though j as

$$r(i, j) = (L - w(i, j))^2$$

Write a program to read paragraphs of text and to lay them out in a way that no line contains more than L characters, for a specified L, and so that you minimize the total raggedness added up over all lines except the last one. (The final line of a paragraph can be arbitrarily shorter then the lines above it.) Line terminator characters are not counted as part of the line width.

Input

Input will consist of one or more datasets.

Each dataset begins with a line containing one integer, L, denoting the maximum line width (not counting line terminator characters). You are guaranteed that $0 < L \leq 80$. A value of zero indicates the end of input.

The remainder of the dataset consists of up to 250 lines containing a paragraph of text, terminated by an empty line. Paragraphs may contain from 1 to 500 words, where a word is any consecutive sequence of non-whitespace characters.

No line of text will contain a word of length greater than L.

Output

Print each paragraph laid out optimally as described above. After each paragraph print a line containing "===" (three equal signs).

If there is more than one way to fill a paragraph with the optimal raggedness, any such layout may be printed.

Example

Given the input

```
6
See if we
care.

25
Raggedy, raggedy are we.
Just as raggedy as raggedy can be.
We don't get nothin' for our labor.
So raggedy, raggedy are we.
- P Seeger

0
```

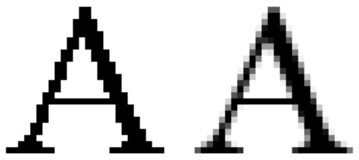
the output would be

```
See
if we
care.
===
Raggedy, raggedy are
we. Just as raggedy
as raggedy can be. We
don't get nothin' for
```

```
our labor. So raggedy,
raggedy are we. - P
Seeger
===
```

Problem C: AAAA

Forays into Anti-Aliased Ascii Art



Aliasing is the term used for artifacts introduced when digitally sampling an analog source due to the finite resolution of the digital capture. Aliasing is a common problem in computer graphics, where lines and smooth curves appear jagged when plotted as pixels. For example, given an equation of a line to be plotted:

$$y = mx + b$$

a naive attempt to draw this line might result in something like

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

looking more like a staircase than a smooth line.

Aliasing (a.k.a. *jaggies*) can be countered by *anti-aliasing* schemes in which pixels are drawn in varying shades of gray and, sometimes, diffused over neighboring pixels to yield a smoother-looking image when viewed from sufficient distance, as illustrated by the picture at the top of this page.

One scheme for anti-aliasing lines works on the idea of shading two pixels at a time for each value for each x. Given a point (x, y) where $y = mx + b$, let y_w be the "whole part" of y (the largest integer that is less than or equal to y) and let y_f be the "fractional part" of y such that

$$y_w + y_f = y$$

For example, if $y = 23.56$, then $y_w = 23$ and $y_f = 0.56$. Let the *gray level* of a pixel be a number from 0.0 to 1.0 where 0.0 denotes a pure white pixel and 1.0 denotes a pure black pixel. If y_f is zero, then shade the pixel (x, y_w) at a gray level of 1.0. If y_f is non-zero, then shade the pixel (x, y_w) at a gray level of $1 - y_f$ and shade the pixel $(x, y_w + 1)$ at a gray level of y_f.

Write a program to draw anti-aliased lines according to this scheme.

Input

The input set will consist of several cases. Each case is given as a single line, containing two numbers, m and b, denoting the slope and intercept of the line in the formula

$$y = mx + b$$

These numbers will be presented as floating point numbers with no more than 2 digits after the decimal point. m will be in the range 0.00 to 0.50 inclusive and b will be in the range -20.00 to 20.00 inclusive.

A zero value for both m and b signals the end of input and is not plotted.

Output

For each line in the input, produce a separate plot consisting of a 20x20 square of characters. Each character represents a point on the portion of the Cartesian plane defined by $0 \leq x < 20$, $0 \leq y < 20$. Each character position is filled with a character obtained by computing the appropriate gray level as described above, rounding to the closest tenth, and then selecting a character from the following table:

Rounded gray scale:	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Character:	.	:	-	=	+	t	w	*	#	%	@

The characters, if you have difficulty recognizing them, are: period, colon, hyphen, equals, plus, lower-case T, lower-case W, asterisk, hash, percent, at.

Each line of the plot will be printed as 20 characters, immediately preceded and immediately followed by a vertical bar (‘|’).

After each plot, print a single line containing 22 underscore (‘_’) characters.

Example

Given the input

```
0.5 0
0.2 10.0
0 0
```

the output would be

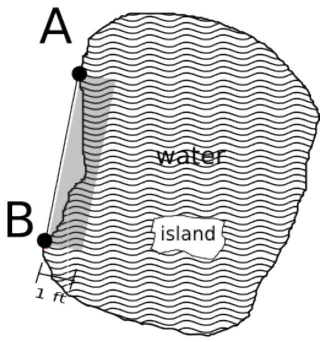
```
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                   t|
|                 t@t|
|               t@t  |
|             t@t    |
|           t@t      |
|         t@t        |
|       t@t          |
|     t@t            |
|   t@t              |
| t@t                |
|@t                  |
|                    |
|                    |
|                    |
|                    |
|                    |
|                -+w#|
|           -+w#@#w+-|
|      -+w#@#w+-     |
| -+w#@#w+-          |
|@#w+-               |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
```

Problem D: Thunk and Plunk

A group of spelunkers are gathered at the top of a deep vertical shaft in a cave. They have enough rope to lower themselves down, but one of them has tossed a small stone down the shaft and heard the "plunk" of a stone landing in water. They don't want to go down the shaft if the water there is deep, as there is then likely to be no exit from the bottom of the shaft that does not require swimming, and they are not equipped for diving in caves.

They continue to drop stones down the shaft, listening for the "thunk" or "plunk" sounds of a stone hitting solid ground or water. They quickly establish that there is a mixture of both at the bottom of the shaft. But if the area covered by water is unbroken, they believe it is likely that the water is quite deep, though perhaps surrounded by a solid shelf. On the other hand, if there are "islands" of solid ground poking out of the water, then the water overall is unlikely to be deep and the shaft may be worth exploring.

Given the locations where they have dropped the stones and the sound they have heard for each, determine if there is definite evidence of solid ground completely surrounded by water.



Assume that the outer perimeter of the water-covered area is sharply defined (i.e., there is no confusion about what is water and what is land) and is "reasonably" smooth: all points on an island are at least one foot inside the the perimeter of the water, and for any two points A and B on the outer perimeter of the water such that a line segment between them does not touch water between A and B, the ground of the outer shore can extend inward no more than one foot inside the line AB.

Input

The input consists of several test cases. Each case begins with a line containing a single integer N, indicating the number of stones dropped. For each case, $3 \leq N \leq 200$. A zero value indicates the end of input. Following that line, there will be N lines, each containing

$$x \; y \; s$$

where x and y are floating point numbers in the range $-100.0 \ldots 100.0$ and s is a single character, either 'T' for "thunk" or 'P' for "plunk". x and y give the coordinates, measured in feet, of the place where the stone landed. There will be at least two "plunk"s in each test case.

Output

For each test case, print a single line containing either the phrase

```
There must be an island.
```

if there is at least one "thunk" point that is definitely completely surrounded by water, or

```
There might not be an island.
```

if no such point exists.

Example

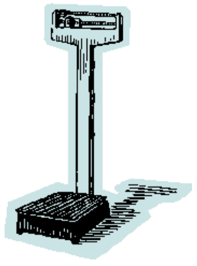
Given the input

```
8
0 0 P
1.0 7.0 T
6.0 0 P
0 6.0 P
6.0 6.0 P
2.5 4.0 P
4.0 4.0 T
4.0 2.0 P
8
1.0 7.0 T
6.0 0 P
0 6.0 P
6.0 6.0 P
2.5 4.0 P
5.5 4.0 T
4.0 2.0 P
0
```

the output would be

```
There must be an island.
There might not be an island.
```

Problem E: Losers Are Winners



Inspired by a popular TV reality show, a local Health and Nutrition Counseling Center is staging a contest to encourage overweight couples to lose weight. A prize will be awarded to the two-person team who manages to lose the highest percentage of weight their combined initial weight over the course of the contest.

For example:

Contestant			Team			
Name	Initial Weight	Final Weight	Name	Initial Weight	Final Weight	% lost
Amanda	220.6	162.5	Green			
Ernesto	320.9	231.0	Green	541.5	393.5	27.3%
Paula	212.0	155.1	Orange			
Eduardo	240.9	170.1	Orange	452.9	325.2	28.2%

In this example, team Orange would be the winner.

Write a program to process the weight loss data and to declare the contest winner.

Input

Input will consist of multiple datasets. Each dataset describes a single contest.

The first line in each dataset consists of a positive integer, N, which represents the number of teams in that contest ($1 \leq N \leq 100$). A value of zero denotes the end of the input.

This is followed by $2N$ lines, each line describing one contestant. The line contains four items: the name of the contestant, followed by the name of the team that contestant is on, followed by two floating point numbers in the range 50.0 to 500.0. The first number represents the contestant's initial weight and the second number represents the final weight at the end of the contest. Contestant names and team names are comprised of alphabetic characters only, with no embedded blanks. Team names will be unique. Contestant names are not guaranteed to be unique.

Output

For each dataset, produce one line of output. That line will contain the name of the winning team, a single blank, then the percentage weight lost by that team printed to one decimal precision, followed immediately by a percent sign.

If two or more teams tie for the greatest percentage weight lost, then list the team whose team name comes earliest when listed in alphabetic order.

Example

Given the input

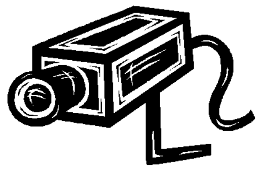
```
2
Amanda Green 220.6 162.5
Paula Orange 212.0 155.1
```

```
Eduardo Orange 240.9 170.1
Ernesto Green 320.9 231.0
6
Carlos White 243.9 176.8
Kandy Red 275.7 208.2
Amanda Green 220.6 162.5
Paula Orange 212.0 155.1
Fred Brown  300.9 210.7
Nadia Pink 217.8 155.9
Ray Red 254.8 166.2
Curtis White 321.1 216.2
Adrian Brown 284.6 201.2
Kara Pink 233.1 176.6
Eduardo Orange 240.9 170.1
Ernesto Green 320.9 231.0
0
```

the output would be

```
Orange 28.2%
White 30.4%
```

Problem F: Line of Sight



The factory foreman at Widgets Inc. is concerned that whenever he leaves the factory floor to go back to his office to do paperwork the workers on the factory floor start goofing off.

Management has decided to install a surveillance camera in the factory so that the foreman can monitor the workers from his office. They are looking at a new type of camera that is mounted on a rail that runs the entire length of one wall. The camera slides back and forth along this rail, while also panning (rotating its angle of view) back-and-forth. Consequently, it can see anything that could be seen by an observer standing anywhere along that wall.

When the camera's installation crew arrived, one of them noted that the large widget-stamping machine on the factory floor might block the view of the camera. The foreman is concerned that some workers might hide in the "shadow" of the widget-stamper to avoid being seen. The camera salesman would like to assure the foreman that the occluded area is simly not large enough for this to be a real problem.

Given the dimensions of the warehouse and the location and dimension of the widget-stamping machine, determine what percentage of the open floor space of the factory (the area not occupied by the widget-stamping machine) would be hidden from the camera.

Input

Input consists of several cases. Each case is presented on a single line of input, which contains 6 floating point numbers:

$$w_1\ h_1\ w_2\ h_2\ x\ y$$

w_1 and h_1 describe the dimensions of the rectangular factory floor. The camera rail is mounted along one of the walls measured by w_1. You are guaranteed that $0 < w_1 \leq 10000.0$ and $0 < h_1 \leq 10000.0$.

w_2 and h_2 describe the dimensions of the rectangular widget-stamping machine. You are guaranteed that $0 < w_2 < w_1$ and $0 < h_2 \leq h_1$. In other words, the machine does fit in the factory.

x describes the distance of the center of the widget-stamping machine from one of the walls measured by h_1. y is the distance of the center of the machine from the wall holding the camera rail. You are guaranteed that $0 < x - \frac{w_2}{2}$, $x + \frac{w_2}{2} < w_1$, $0 < y - \frac{h_2}{2}$, and $y + \frac{h_2}{2} < h_1$.

End of input is signaled by a line of six numbers of which the first is zero.

Output

For each test case, print a single line of output in the form

```
P%
```

where P is a floating point number in the range $0 \leq P \leq 100$, denoting the percentage of the open floor space of the factory that is hidden from the camera. P should be printed to a precision of one decimal digit.

Example

Given the input

```
100 100 10 10 50 50
100.0 100.0 25 25 50.0 50
0 0 0 0 0 0
```

the output would be

```
0.3%
2.8%
```

Problem G: Stained Glass

Dale was working on the restoration of an elaborate stained glass window composed of numerous irregular pieces of colored glass. He had carefully disassembled the window, treated the more faded pieces of glass to restore their bright coloring, and had made substantial progress towards reassembling the work when night fell and the poor lighting forced him to stop work.

Upon returning the next morning, he discovered that some well-meaning janitor had disposed of his drawings and pictures of the original window. Now he has to figure out how to arrange the pieces to fit the hole remaining in the window. Dale knows that, because of the techniques used to prepare ancient stained glass, the glass was not of uniform thickness but was always cut and arranged so that the thickest edge would be at the bottom. He therefore knows the vertical orientation of each piece (they need not be rotated) but not the facing (it may be necessary to flip some pieces horizontally).

Input

Input consists of multiple datasets, terminated by a line containing the left-justified string "***".

Each dataset consists of silhouettes of one or more pieces and a silhouette of the hole in the window. A silhouette is presented as an ASCII graphic composed of blanks and a specific non-blank character (as described further below). A blank character indicates locations where no glass exists. The non-blank characters indicate the presence of glass (or, in the case of the hole's silhouette, a location where we wish to place some glass).

Each silhouette consists of 1 to 8 lines, each line containing 1 to 8 characters. Each line of a silhouette will have at least one non-blank character. Each silhouette is presented in a separate sequence of lines.

The first piece is rendered using the character 'A', then the next piece with a 'B', and so on. There will be at most 8 pieces. After the final piece, the silhouette of the hole is rendered using the character '#'. The transition from one silhouette to the next is signaled by the change of character. The end of the hole's silhouette and of the dataset is signaled by an empty line.

Output

For each dataset, print a line containing ===== (5 equal signs). Then, if an arrangement of the pieces that exactly and completely fills the holes silhouette is possible, print that silhouette filled in with the alphabetic characters indicating the positioning of the pieces. (If more than one arrangement is possible, you may show any one of them.) The filled-in silhouette should be printed as far to the left as possible without distorting the shape of the silhouette. There should be no blanks spaces at the ends of the output lines.

If no such arrangement is possible, print "The window cannot be repaired.".

Example

Given the input

```
A
AAA
A
A
AAA
B
BBB
  B
  B
 BB
  B
    CC
     C
  #
 ####
 ####
 ####
 ####
 ####

A
 B
  C
##
##

***
```

the output would be

```
=====
 B
ABBB
AAAB
ACCB
ACBB
AAAB
=====
The window cannot be repaired.
```

Problem H: Road Rally

Consider a race track laid out on a rectangular grid, like this:

```
xxxxXxxxxXxxxxXxxxxXxxxxXxxxxX
xxxxxx      xxxxx      xxx  xx
xxxxx        xxx        xx    x
xx     xx     x        x     x
X      xx        2     x  x  x
x      xx                 x  x
x      xxxxxxxxxxxxxx     x  x
x      xxxxxxxxxxxxxxxxxxxx  x
x  3     xxxxxxxxxxxxxx      x
X          xxxxxxxx          x
x                            x
x  xxx                xxx 1  x
x  xxx       0         x     x
x                            x
Xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

An ‘x’ or ‘X’ denotes a wall or barrier of some kind. Each numeric digit indicates a checkpoint. A motorcycle starts at checkpoint ‘0’ and must visit each numbered checkpoint in order, ending the course at the highest-numbered checkpoint.

Movement of the motorcycle is constrained as follows: In the first second, the motorcycle must move to one of the 8 neighbors of its starting position. Each second after that, the motorcycle may move to a point P offset by the same horizontal and vertical distance by which the motorcycle moved in the previous second, or it may move to any of the eight neighbors of that point P. The motorcycle may not, however, land outside the grid or on an ‘x’ or ‘X’. It may, however, leap over one or more ‘x’s or ‘X’s as long as it lands in an empty square or at a checkpoint.

For example, in the layout above, the rider might move according to the sequence shown as a, b, c, . . . as shown below:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxx      xxxxx      xxx  xx
xxxxx        xxx       xx    x
xx     xx     x        x     x
x      xx        2     x  x  x
x      xx                 x  x
x      xxxxxxxxxxxxxx     x  x
x      xxxxxxxxxxxxxxxxxxxx  x
x  3     xxxxxxxxxxxxxx      x
x          xxxxxxxx          x
x                            x
x  xxx                xxx 1  x
x  xxx       0a b      xe    x
x                  c  d      x
xxxxXxxxxXxxxxXxxxxXxxxxXxxxxX
```

and then on to the first checkpoint in 6 seconds.

Write a program to find the shortest time required to start at '0' and end at the final checkpoint, landing upon each intervening checkpoint in the order indicated by the digits. The motorcycle may land on a checkpoint out of order while on its way to another checkpoint, but does not receive credit for having visited that checkpoint unless it has already visited all of the lower-numbered checkpoints.

Input

Input consists of multiple race courses. Each course begins with a line containing two integers, w, and h, denoting the width and height of the track. You are guaranteed that $1 \leq w \leq 40$ and $1 \leq h \leq 40$. A zero value for w and h denoted the end of the input.

This is followed by h lines, each line containing w characters. These define the race track as explained above. Each track will have an area of at least 2 and will contain at least two checkpoints (0 and 1) and no more than 10 checkpoints. Where more than two checkpoints are presented, the checkpoints will be in strict sequence - there will be no "gaps" in the numbering.

Output

Print the minimum # of seconds required to complete each course as an integer, one per line. If it is not possible to complete a course, print -1 instead.

Example

Given the input

```
5 5
xxxxx
x 0 x
x 12x
x   x
xxxxx
10 3
xxxxxxxxxx
x 0x  1  x
xxxxxxxxxx
10 2
xxxxxxxxxx
x 0xx 1  x
30 15
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxx      xxxxx      xxx  xx
xxxxx        xxx       xx    x
xx     xx     x        x     x
x      xx        2     x  x  x
x      xx                 x  x
x      xxxxxxxxxxxxxx     x  x
x      xxxxxxxxxxxxxxxxxxxx  x
x  3     xxxxxxxxxxxxxx      x
x          xxxxxxxx          x
x                            x
x  xxx                xxx 1  x
x  xxx       0         x     x
x                            x
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0 0
```

the output would be

```
2
5
-1
16
```