Problem A: Parity

Source file: `parity.{c, cpp, java}`
Input file: `parity.in`

A bit string has *odd parity* if the number of 1's is odd. A bit string has *even parity* if the number of 1's is even. Zero is considered to be an even number, so a bit string with no 1's has even parity. Note that the number of 0's does not affect the parity of a bit string.

Input: The input consists of one or more strings, each on a line by itself, followed by a line containing only "#" that signals the end of the input. Each string contains 1–31 bits followed by either a lowercase letter 'e' or a lowercase letter 'o'.

Output: Each line of output must look just like the corresponding line of input, except that the letter at the end is replaced by the correct bit so that the entire bit string has even parity (if the letter was 'e') or odd parity (if the letter was 'o').

Example input:	Example output:
101e 0100100 1e 000e 110100101o #	1010 0100101 11 0000 1101001010

Last modified on October 7, 2008 at 5:28 PM.

(this page intentionally left blank)

Problem B: Lampyridae Teleportae

Source file: `firefly.{c, cpp, java}`
Input file: `firefly.in`

The discovery of a remarkable new insect, the Lampyridae Teleportae, more commonly known as the teleporting firefly, has sparked a no-less-remarkable number of ways to try to catch them. Rather than flying, the Lampyridae Teleportae teleports from spot to spot by warping space-time. When it stops between teleports, it hovers for a moment and flashes its light in search of a mate. So, even though they only come out after dark, it's easy to observe them, but very difficult to catch them. Fortunately for the Association for Catching Lampyridae (ACL), student members of the Association for Cool Machinery (ACM) recently developed the world's first teleporting tennis shoes. The tennis shoes are efficient enough that, when a Lampyridae Teleportae is spotted by its flash, there is always time to teleport once before the firefly itself teleports off to another location, but there is never time to teleport twice in a row before the firefly teleports away. The tennis shoes have a maximum teleport range, however, depending on how well their flux capacitor is constructed, so it's not always possible to catch a Lampyridae Teleportae with just a single teleport. The most efficient catching method is to remain in place until a firefly flashes, and to then teleport in a straight line directly toward it, subject to the limitation of the maximum range of ones tennis shoes, in an attempt to get close enough to catch it. If you don't get close enough, you wait for the next flash, teleport towards it again, and repeat, until you either catch it or it's gone.

For this programming problem you will simulate this procedure for catching teleporting fireflies with a few simplifying assumptions: (1) We will be chasing only one firefly at a time. (2) Firefly chasing will take place in two dimensions where all units are considered to be yards. (3) The firefly is "caught" if the chaser can manage to come within one yard of the firefly. (4) The chaser's movement toward a firefly is always in a straight line from his or her current location directly toward the flash; if the range of the chaser's tennis shoes prevents getting close enough to catch the firefly, the chaser will always teleport the maximum range possible (thus, although the chaser always starts at integer coordinates, it is possible and likely that any or all of the chaser's locations after the first teleport will be at non-integer coordinates).

The input will consist of several chase scenarios. For each scenario you will be given the maximum range in yards of the chaser's teleporting tennis shoes, the chaser's starting location, and a list of one or more flash locations for the firefly being chased. For each chase scenario your program will output a single line indicating either the flash location where the firefly was caught, or a message noting that the firefly was never caught.

Input: The first line of a chase scenario contains three numbers, delimited by a single space, in the following order: the maximum range in yards of the chaser's teleporting tennis shoes, the starting x-coordinate of the chaser, and the starting y-coordinate of the chaser. The maximum range will be a positive integer from 1 to 1000. The x and y values for the starting coordinates will be integers from 0 to 1000. The remaining lines of an input scenario contain two integers each, an x-coordinate and a y-coordinate, again delimited by a single space. These are, in order of appearance, the locations where the firefly flashes. All coordinate values range from 0 to 1000. A line specifying a value of -1 for both x and y terminates the list, at which point we consider the firefly to disappear never to be seen again. Note that a firefly might be caught at a flash location prior to end of the list; in this case the rest of the flash locations listed in the input for the current chase scenario should simply be ignored.

The next input scenario begins on the line immediately after the last line of the preceding scenario. An input scenario that specifies 0 (zero) as the maximum range of the chaser will terminate the input.

Output: Every output line will be either: (1) "Firefly N caught at (x,y)", where N is the input scenario number starting with 1, and (x,y) is the last location the firefly flashed before it was caught; or (2) "Firefly N not caught".

Example Input:	Example Output:
2 0 0 3 3 4 4 5 5 6 6 7 7 -1 -1 2 0 0 3 3 5 5 7 7 -1 -1 10 50 50 50 62 40 55 30 55 45 45 50 50 55 55 50 50 -1 -1 0 0 0	Firefly 1 caught at (6,6) Firefly 2 not caught Firefly 3 caught at (50,50)

Last modified on October 7, 2008 at 1:56 PM.

Problem C: Hex Tile Equations

Source file: `hex.{c, cpp, java}`
Input file: `hex.in`

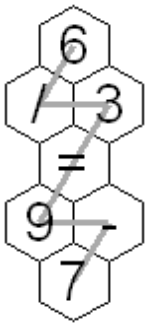


Figure 1

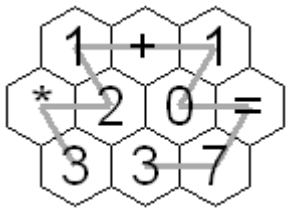


Figure 2

An amusing puzzle consists of a collection of hexagonal tiles packed together with each tile showing a digit or '=' or an arithmetic operation '+', '-', '*', or '/'. Consider continuous paths going through each tile exactly once, with each successive tile being an immediate neighbor of the previous tile. The object is to choose such a path so the sequence of characters on the tiles makes an *acceptable* equation, according to the restrictions listed below. A sequence is illustrated in each figure above. In Figure 1, if you follow the gray path from the top, the character sequence is"6/3=9-7". Similarly, in Figure 2, start from the bottom left 3 to get "3*21+10=73".

There are a lot of potential paths through a moderate sized hex tile pattern. A puzzle player may get frustrated and want to see the answer. Your task is to automate the solution.

The arrangement of hex tiles and choices of characters in each puzzle satisfy these rules:

1. The hex pattern has an odd number of rows greater than 2. The odd numbered rows will all contain the same number of tiles. Even numbered rows will have one more hex tile than the odd numbered rows and these longer even numbered rows will stick out both to the left and the right of the odd numbered rows.
2. There is exactly one '=' in the hex pattern.
3. There are no more than two '*' characters in the hex pattern.
4. There will be fewer than 14 total tiles in the hex pattern.
5. With the restrictions on allowed character sequences described below, there will be a unique acceptable solution in the hex pattern.

To have an acceptable solution from the characters in some path, the expressions on each side of the equal sign must be in acceptable form and evaluate to the same numeric value. The following rules define acceptable form of the expressions on each side of the equal sign and the method of expression evaluation:

6. The operators '+', '-', '*', and '/' are only considered as binary operators, so no character sequences where '+' or '-' would be a unary operator are acceptable. For example "-2*3=-6" and "1 =5+-4" are not acceptable.
7. The usual precedence of operations is not used. Instead all operations have equal precedence and operations are carried out from left to right. For example "44-4/2=2+3*4" is acceptable and "14=2+3*4" is not acceptable.

8. If a division operation is included, the equation can only be acceptable if the division operation works out to an exact integer result. For example "10/5=12/6" and "7+3/5=3*4/6" are acceptable. "5/2*4=10" is not acceptable because the sides would only be equal with exact mathematical calculation including an intermediate fractional result. "5/2*4=8" is not acceptable because the sides of the equation would only be equal if division were done with truncation.
9. At most two digits together are acceptable. For example, "123+1 = 124" is not acceptable.
10. A character sequences with a '0' directly followed by another digit is not acceptable. For example, "3*05=15" is not acceptable.

With the assumptions above, an acceptable expression will never involve an intermediate or final arithmetic result with magnitude over three million.

Input: The input will consist of one to fifteen data sets, followed by a line containing only 0.

The first line of a dataset contains blank separated integers *r c*, where *r* is the number of rows in the hex pattern and *c* is the number of entries in the odd numbered rows. The next *r* lines contain the characters on the hex tiles, one row per line. All hex tile characters for a row are blank separated. The lines for odd numbered rows also start with a blank, to better simulate the way the hexagons fit together. Properties 1-5 apply.

Output: There is one line of output for each data set. It is the unique acceptable equation according to rules 6-10 above. The line includes no spaces.

Example input:	Example output:
5 1 6 / 3 = 9 - 7 3 3 1 + 1 * 2 0 = 3 3 7 5 2 9 - * 2 = 3 4 + 8 3 4 / 0	6/3=9-7 3*21+10=73 8/4+3*9-2=43

Last modified on October 26, 2008 at 9:27 PM.

Problem D: The Bridges of San Mochti

Source file: `bridges.{c, cpp, java}`
Input file: `bridges.in`

You work at a military training facility in the jungles of San Motchi. One of the training exercises is to cross a series of rope bridges set high in the trees. Every bridge has a maximum capacity, which is the number of people that the bridge can support without breaking. The goal is to cross the bridges as quickly as possible, subject to the following tactical requirements:

One unit at a time!

If two or more people can cross a bridge at the same time (because they do not exceed the capacity), they do so as a unit; they walk as close together as possible, and they all take a step at the same time. It is never acceptable to have two different units on the same bridge at the same time, even if they don't exceed the capacity. Having multiple units on a bridge is not tactically sound, and multiple units can cause oscillations in the rope that slow everyone down. This rule applies even if a unit contains only a single person.

Keep moving!

When a bridge is free, as many people as possible begin to cross it as a unit. Note that this strategy doesn't always lead to an optimal overall crossing time (it may be faster for a group to wait for people behind them to catch up so that more people can cross at once). But it is not tactically sound for a group to wait, because the people they're waiting for might not make it, and then they've not only wasted time but endangered themselves as well.

Periodically the bridges are reconfigured to give the trainees a different challenge. Given a bridge configuration, your job is to calculate the minimum amount of time it would take a group of people to cross all the bridges subject to these requirements.

For example, suppose you have nine people who must cross two bridges: the first has capacity 3 and takes 10 seconds to cross; the second has capacity 4 and takes 60 seconds to cross. The initial state can be represented as (9 0 0), meaning that 9 people are waiting to cross the first bridge, no one is waiting to cross the second bridge, and no one has crossed the last bridge. At 10 seconds the state is (6 3 0). At 20 seconds the state is (3 3 /3:50/ 0), where /3:50/ means that a unit of three people is crossing the second bridge and has 50 seconds left. At 30 seconds the state is (0 6 /3:40/ 0); at 70 seconds it's (0 6 3); at 130 seconds it's (0 2 7); and at 190 seconds it's (0 0 9). Thus the total minimum time is 190 seconds.

Input: The input consists of one or more bridge configurations, followed by a line containing two zeros that signals the end of the input. Each bridge configuration begins with a line containing a negative integer –B and a positive integer P, where B is the number of bridges and P is the total number of people that must cross the bridges. Both B and P will be at most 20. (The reason for putting –B in the input file is to make the first line of a configuration stand out from the remaining lines.) Following are B lines, one for each bridge, listed in order from the first bridge that must be crossed to the last. Each bridge is defined by two positive integers C and T, where C is the capacity of the bridge (the maximum number of people the bridge can hold), and T is the time it takes to cross the bridge (in seconds). C will be at most 5, and T will be at most 100. Only one unit, of size at most C, can cross a bridge at a time; the time required is always T, regardless of the size of the unit (since they all move as one). The end of one bridge is always close to the beginning of the next, so the travel time between bridges is zero.

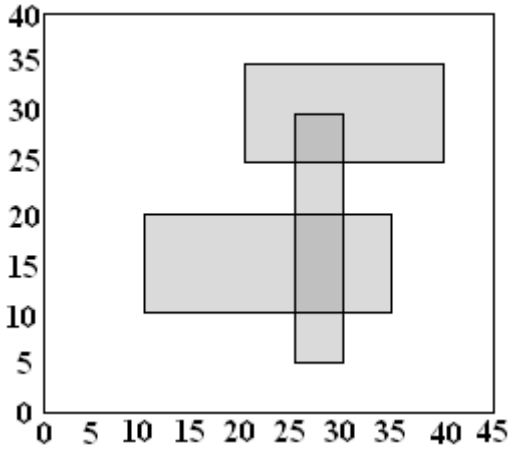
Output: For each bridge configuration, output one line containing the minimum amount of time it will take (in seconds) for all of the people to cross all of the bridges while meeting both tactical requirements.

Example input:	Example output:
-1 2 5 17 -1 8 3 25 -2 9 3 10 4 60 -3 10 2 10 3 30 2 15 -4 8 1 8 4 30 2 10 1 12 0 0	17 75 190 145 162

Last modified on October 22, 2008 at 10:24 PM.

Problem E: Bulletin Board

Source file: `bulletin.{c, cpp, java}`
Input file: `bulletin.in`



The ACM Student Chapter has just been given custody of a number of school bulletin boards. Several members agreed to clear off the old posters. They found posters plastered many levels deep. They made a bet about how much area was left clear, what was the greatest depth of posters on top of each other, and how much of the area was covered to this greatest depth. To determine each bet's winner, they made very accurate measurements of all the poster positions as they removed them. Because of the large number of posters, they now need a program to do the calculations. That is your job.

A simple illustration is shown above: a bulletin board 45 units wide by 40 high, with three posters, one with corners at coordinates (10, 10) and (35, 20), another with corners at (20, 25) and (40, 35), and the last with corners at (25, 5) and (30, 30). The total area not covered by any poster is 1300. The maximum number of posters on top of each other is 2. The total area covered by exactly 2 posters is 75.

Input: The input will consist of one to twenty data sets, followed by a line containing only 0. On each line the data will consist of blank separated nonnegative integers.

The first line of a dataset contains integers *n w h*, where *n* is the number of posters on the bulletin board, *w* and *h* are the width and height of the bulletin board. Constraints are $0 < n \leq 100$; $0 < w \leq 50000$; $0 < h \leq 40000$.

The dataset ends with n lines, each describing the location of one poster. Each poster is rectangular and has horizontal and vertical sides. The x and y coordinates are measured from one corner of the bulletin board. Each line contains four numbers *xl yl xh* and *yh*, where *xl* and *yl*, are the lowest values of the x and y coordinates in one corner of the poster and *xh* and *yh* are the highest values in the diagonally opposite corner. Each poster fits on the bulletin board, so $0 \leq xl < xh \leq w$, and $0 \leq yl < yh \leq h$.

Output: There is one line of output for each data set containing three integers, the total area of the bulletin board that is not covered by any poster, the maximum depth of posters on top of each other, and the total area covered this maximum number of times.

Caution: An approach examining every pair of integer coordinates might need to deal with 2 billion coordinate pairs.

Example input:	Example output:
3 45 40 10 10 35 20 20 25 40 35 25 5 30 30 1 20 30 5 5 15 25 2 2000 1000 0 0 1000 1000 1000 0 2000 1000 3 10 10 0 0 10 10 0 0 10 10 0 0 10 10 0	1300 2 75 400 1 200 0 1 2000000 0 3 100

Last modified on October 26, 2008 at 10:00 PM.

Problem F: Serial Numbers

Source file: `serials.{c, cpp, java}`
Input file: `serials.in`

A manufacturer keeps an ordered table of serial numbers by listing in each row of the table a range of serial numbers along with two corresponding pieces of information called the status code and the transfer code. A four-column table stores information about ranges of serial numbers in this order: starting serial number, ending serial number, status code, transfer code. Serial numbers as well as transfer codes are integers from 1 to 2^{31}-1 (2^{31}-1 = 2147483647), and status codes are a single upper-case letter. The table is maintained in increasing order of serial numbers, serial number ranges are never allowed to overlap, and for any given serial number, the table must always accurately represent the most recent data (status code and transfer code) for that serial number.

Let's say that 100,000 serial numbers are created with a status of "A" and a transfer code of "1". An entry for those serial numbers might look like this:

```
1 100000 A 1
```

This is obviously far more efficient than storing 100,000 individual rows all with the same status and transfer codes. The challenge arises when serial numbers within already defined ranges need to be given different status or transfer codes. For example, if serial number 12345 needs to change to status B, the above table would need to become three separate entries:

```
1 12344 A 1
12345 12345 B 1
12346 100000 A 1
```

Now let's change the transfer code of all serial numbers in the range 12000 to 12999 to 2. This gets us:

```
1 11999 A 1
12000 12344 A 2
12345 12345 B 2
12346 12999 A 2
13000 100000 A 1
```

Now change all existing serial numbers from 10000 to 100000 to status C and transfer code 2:

```
1 9999 A 1
10000 100000 C 2
```

Once created a serial number will never be deleted, but it is possible to have ranges of undefined serial numbers between ranges of defined ones. To demonstrate, let's now set all serial numbers from 1000000 to 1999999 to status Z and transfer code 99:

```
1 9999 A 1
10000 100000 C 2
1000000 1999999 Z 99
```

Finally, the table is always maintained with a minimal number of rows, meaning specifically that there will never be two adjacent rows in the table where one would suffice. For example, consider the following serial number table:

```
1 10 A 1
11 20 A 1
21 30 B 1
```

The first two rows could actually be represented by a single row, meaning that the table above does not have a minimal number of rows. The same data represented by a minimal number of rows would look like this:

```
1 20 A 1
21 30 B 1
```

The following table, however, because the first two rows have non-matching transfer codes, already contains the minimal number of rows:

```
1 10 A 1
11 20 A 2
21 30 B 1
```

Similarly, the following table cannot be reduced further because the first two rows do not represent a continuous series of serial numbers:

```
1 10 A 1
12 20 A 1
21 30 B 1
```

Input: Each input case begins with a single line that is a character string naming the test case. This string contains at most 80 characters. The name "END" marks the end of the input. Following this will be 1 to 100 lines of the form "A B S T", where A, B, and T are integers in the range 1 to 2^{31}-1, S is an uppercase letter, and A<=B. These lines are, in the order they are to be applied, the serial number transactions to be recorded, where A is the start of the serial number range, B is the end of the serial number range, S is the status code, and T is the transfer code. The list of serial number transactions is terminated by a line containing only a 0 (zero) character.

Output: For each input case, echo the test case name to the output on a line by itself, followed by the resulting minimal-rows serial number table that results after all serial number transactions have been applied.

Example Input:	Example Output:
First Example 1 100000 A 1 12345 12345 B 1 0 And Another 1 100000 A 1 12345 12345 B 1 12000 12999 A 2 12345 12345 B 2 0 Test Case Three 1 100000 A 1 12345 12345 B 1 12000 12999 A 2 12345 12345 B 2 10000 100000 C 2 0 Example Four 1 100000 A 1 12345 12345 B 1 12000 12999 A 2 12345 12345 B 2 10000 100000 C 2 1000000 1999999 Z 99 0 Example 5 1 10 A 1 21 30 B 1 11 20 A 1 0 Example 6 21 30 B 1 1 10 A 1 11 20 A 2 0 Example 7 12 20 A 1 21 30 B 1 1 10 A 1 0 END	First Example 1 12344 A 1 12345 12345 B 1 12346 100000 A 1 And Another 1 11999 A 1 12000 12344 A 2 12345 12345 B 2 12346 12999 A 2 13000 100000 A 1 Test Case Three 1 9999 A 1 10000 100000 C 2 Example Four 1 9999 A 1 10000 100000 C 2 1000000 1999999 Z 99 Example 5 1 20 A 1 21 30 B 1 Example 6 1 10 A 1 11 20 A 2 21 30 B 1 Example 7 1 10 A 1 12 20 A 1 21 30 B 1

Last modified on October 26, 2008 at 2:20 PM.

(this page intentionally left blank)

Problem G: Line & Circle Maze

Source file: `maze.{c, cpp, java}`
Input file: `maze.in`

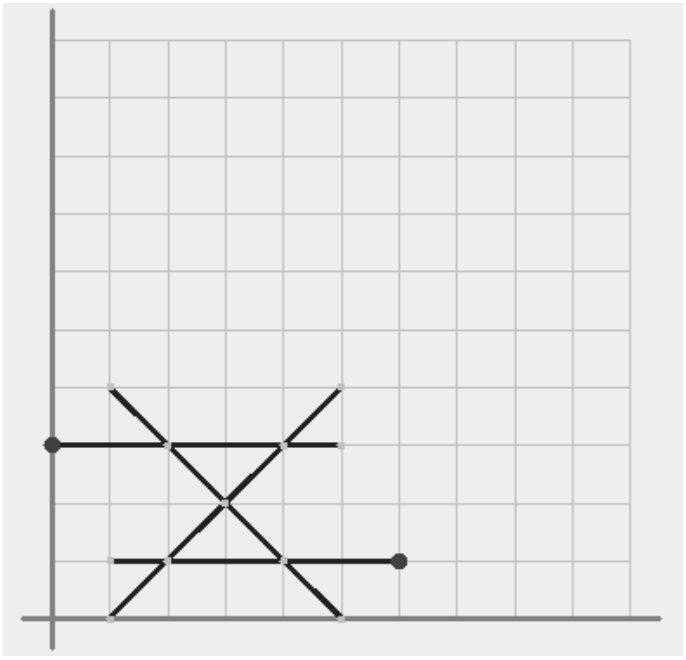
A deranged algorithms professor has devised a terrible final exam: he throws his students into a strange maze formed entirely of linear and circular paths, with line segment endpoints and object intersections forming the junctions of the maze. The professor gives his students a map of the maze and a fixed amount of time to find the exit before he floods the maze with xerobiton particles, causing anyone still in the maze to be immediately inverted at the quantum level. Students who escape pass the course; those who don't are trapped forever in a parallel universe where the grass is blue and the sky is green.

The entrance and the exit are always at a junction as defined above. Knowing that clever ACM programming students will always follow the shortest possible path between two junctions, he chooses the entrance and exit junctions so that the distance that they have to travel is as far as possible. That is, he examines all pairs of junctions that have a path between them, and selects a pair of junctions whose shortest path distance is the longest possible for the maze (which he rebuilds every semester, of course, as the motivation to cheat on this exam is very high).

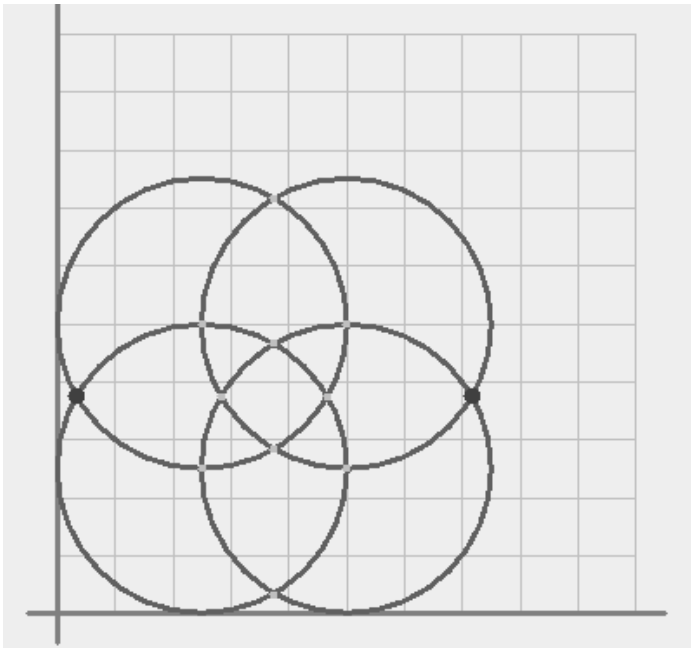
The joy he derives from quantumly inverting the majority of his students is marred by the tedium of computing the length of the longest of the shortest paths (he needs this to know to decide how much time to put on the clock), so he wants you to write a program to do it for him. He already has a program that generates the mazes, essentially just a random collection of line segments and circles. Your job is to take that collection of line segments and circles, determine the shortest paths between all the distinct pairs of junctions, and report the length of the longest one.

The input to your program is the output of the program that generates his mazes. That program was written by another student, much like yourself, and it meets a few of the professor's specifications: 1) No endpoint of a line segment will lie on a circle; 2) No line segment will intersect a circle at a tangent; 3) If two circles intersect, they intersect at exactly two distinct points; 4) Every maze contains at least two junctions; that is, a minimum maze is either a single line segment, or two circles that intersect. There is, however, one bug in the program. (He would like to have it fixed, but unfortunately the student who wrote the code never gave him the source, and is now forever trapped in a parallel universe.) That bug is that the maze is not always entirely connected. There might be line segments or circles, or both, off by themselves that intersect nothing, or even little "submazes" composed of intersecting line segments and circles that as a whole are not connected to the rest of the maze. The professor insists that your solution account for this! The length that you report must be for a path between connected junctions!

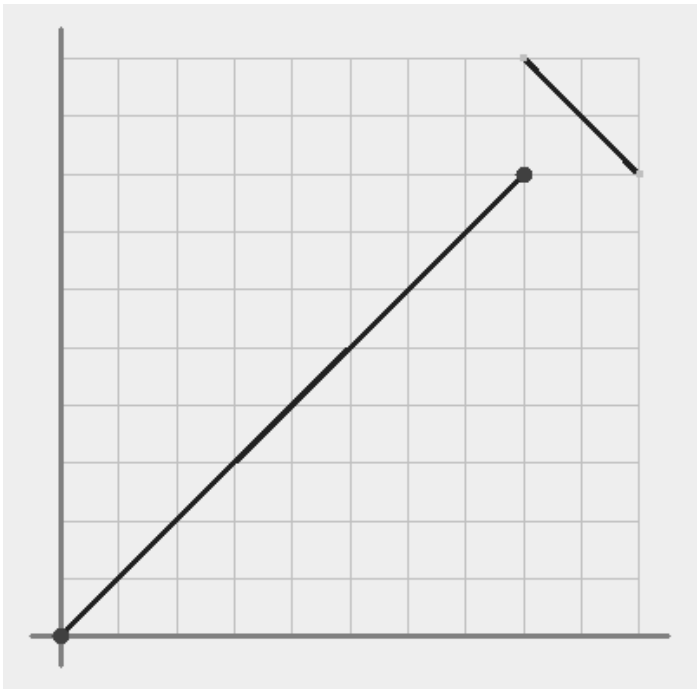
Examples:



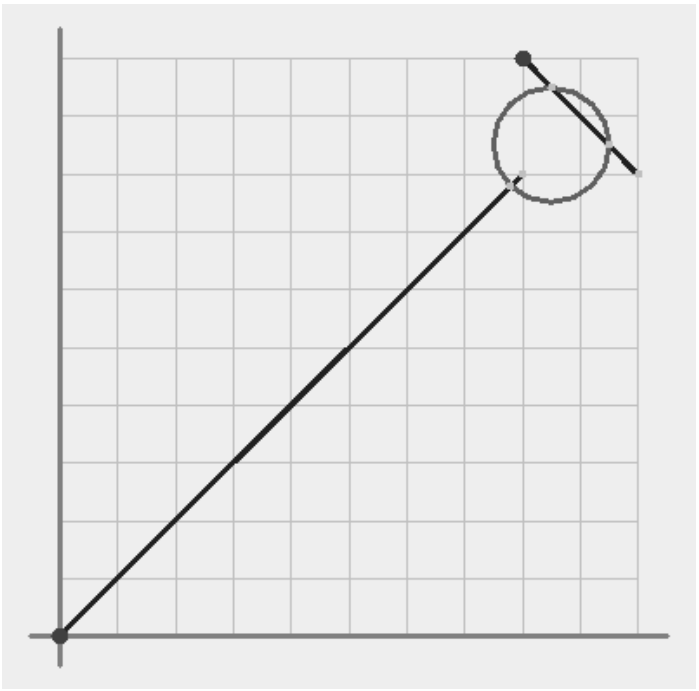
Line segments only. The large dots are the junction pair whose shortest path is the longest possible.



An example using circles only. Note that in this case there is also another pair of junctions with the same length longest possible shortest path.



Disconnected components.



Now the line segments are connected by a circle, allowing for a longer shortest path.

Input: An input test case is a collection of line segments and circles. A line segment is specified as "L X_1 Y_1 X_2 Y_2" where "L" is a literal character, and (X_1,Y_1) and (X_2,Y_2) are the line segment endpoints. A circle is specified by "C X Y R" where "C" is a literal character, (X,Y) is the center of the circle, and R is its radius. All input values are integers, and line segment and circle objects are entirely contained in the first quadrant within the box defined by (0,0) at the lower left and (100,100) at the upper right. Each test case will consist of from 1 to 20 objects, terminated by a line containing only a single asterisk. Following the final test case, a line containing only a single asterisk marks the end of the input.

Output: For each input maze, output "Case N: ", where N is the input case number starting at one (1), followed by the length, rounded to one decimal, of the longest possible shortest path between a pair of connected junctions.

Example Input:	**Example Output:**
L 10 0 50 40 L 10 40 50 0 L 10 10 60 10 L 0 30 50 30 * C 25 25 25 C 50 25 25 C 25 50 25 C 50 50 25 * L 0 0 80 80 L 80 100 100 80 * L 0 0 80 80 L 80 100 100 80 C 85 85 10 * *	Case 1: 68.3 Case 2: 78.5 Case 3: 113.1 Case 4: 140.8

Note: It is recommended that the atan2() function, rather than acos() or asin(), be used when calculating arc angles.

Last modified on October 26, 2008 at 2:15 PM.

Problem H: Steganography

Source file: `steg.{c, cpp, java}`
Input file: `steg.in`

In cryptography, the goal is to encrypt a message so that, even if the the message is intercepted, only the intended recipient can decrypt it. In *steganography*, which literally means "hidden writing", the goal is to hide the fact that a message has even been sent. It has been in use since 440 BC. Historical methods of steganography include invisible inks and tatooing messages on messengers where they can't easily be seen. A modern method is to encode a message using the least-significant bits of the RGB color values of pixels in a digital image.

For this problem you will uncover messages hidden in plain text. The spaces within the text encode bits; an odd number of consecutive spaces encodes a 0 and an even number of consecutive spaces encodes a 1. The four texts in the example input below (terminated by asterisks) encode the following bit strings: 11111, 000010001101101, 01, and 000100010100010011111. Each group of five consecutive bits represents a binary number in the range 0–31, which is converted to a character according to the table below. If the last group contains fewer than five bits, it is padded on the right with 0's.

" " (space)	0
"A" – "Z"	1–26
"'" (apostrophe)	27
"," (comma)	28
"-" (hyphen)	29
"." (period)	30
"?" (question mark)	31

The first message is $11111_2 = 31_{10}$ = "?". The second message is $(00001, 00011, 01101)_2 = (1, 3, 13)_{10}$ = "ACM". The third message is $01\underline{000}_2 = 8_{10}$ = "H", where the underlined 0's are padding bits. The fourth message is $(00010, 00101, 00010, 01111, 1\underline{0000})_2 = (2, 5, 2, 15, 16)_{10}$ = "BEBOP".

Input: The input consists of one or more texts. Each text contains one or more lines, each of length at most 80 characters, followed by a line containing only "*" (an asterisk) that signals the end of the text. A line containing only "#" signals the end of the input. In addition to spaces, text lines may contain any ASCII letters, digits, or punctuation, except for "*" and "#", which are used only as sentinels.

Output: For each input text, output the hidden message on a line by itself. Hidden messages will be 1–64 characters long.

Note: Input text lines and output message lines conform to all of the whitespace rules listed in item 7 of *Notes to Teams* except that there may be consecutive spaces *within* a line. There will be no spaces at the beginning or end of a line.

Example input:	Example output:
Programmer, I would like to see a question mark. * Behold, there is more to me than you might think when you read me the first time. * Symbol for hydrogen? * A B C D E F G H I J K L M N O P Q R S T U V * #	? ACM H BEBOP

Last modified on October 21, 2008 at 9:17 PM.