Problem A: Grade School Multiplication

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

An educational software company, All Computer Math (ACM), has a section on multiplication of integers. They want to display the calculations in the traditional grade school format, like the following computation of 432 × 5678:

```
    432
   5678
-------
   3456
  3024
 2592
2160
-------
2452896
```

Note well that the final product is printed without any leading spaces, but that leading spaces are necessary on some of the other lines to maintain proper alignment. However, as per our regional rules, there should *never* be any lines with *trailing* white space. Note that the lines of dashes have length matching the final product.

As a special case, when one of the digits of the second operand is a zero, it generates a single 0 in the partial answers, and the next partial result should be on the *same* line rather than the next line down. For example, consider the following product of 200001 × 90040:

```
     200001
      90040
-----------
    8000040
180000900
-----------
18008090040
```

The rightmost digit of the second operand is a 0, causing a 0 to be placed in the rightmost column of the first partial product. However, rather than continue to a new line, the partial product of 4 × 200001 is placed on the same line as that 0. The third and fourth least-significant digits of the second operand are zeros, each resulting in a 0 in the second partial product on the same line as the result of 9 × 200001.

As a final special case, if there is only one line in the partial answer, it constitutes a full answer, and so there is no need for computing a sum. For example, a computation of 246 × 70 would be formatted as

```
  246
   70
-----
17220
```

Your job is to generate the solution displays.

Input: The input contains one or more data sets. Each data set consists of two positive integers on a line, designating the operands in the desired order. Neither number will have more than 6 digits, and neither will have leading zeros. After the last data set is a line containing only 0 0.

Output: For each data set, output a label line containing "Problem " with the number of the problem, followed by the complete multiplication problem in accordance with the format rules described above.

Warning: A standard int type cannot properly handle 12-digit numbers. You should use a 64-bit type (i.e., a `long` in Java, or a `long long` in C++).

Example Input:	Example Output:

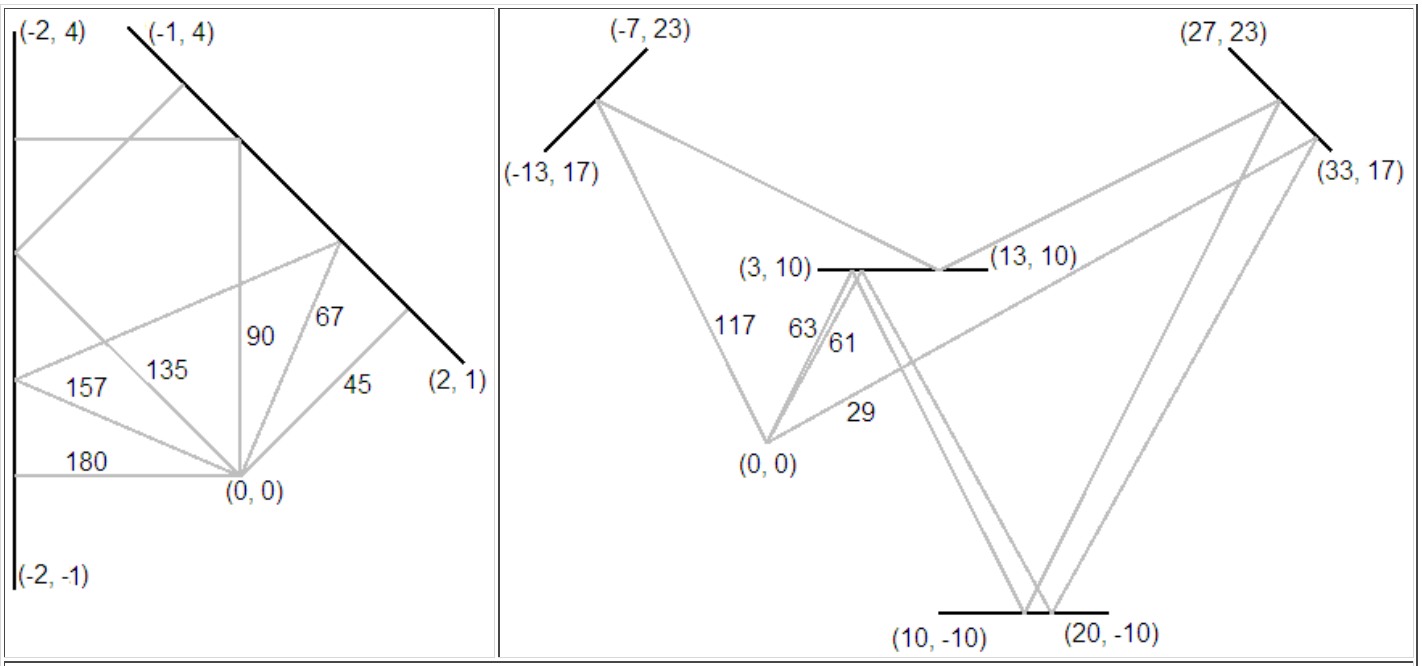
432 5678
200001 90040
246 70
0 0
	Problem 1
	  432
	  5678
	------- 	
	  3456
	  3024
	  2592
	2160
	------- 	
	2452896
	Problem 2
	  200001
	  90040
	----------- 	
	  8000040
	180000900
	----------- 	
	18008090040
	Problem 3
	  246
	  70
	----- 	
	17220

Problem B: Laser Tag

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

A Laser Tag environment is set up with a number of upright, rectangular, double-sided mirrors, all reaching high over head. These mirrors can reflect your laser beam, allowing you multiple ways to shoot your opponents. Unfortunately, they can also reflect your laser beam in such a way that it hits you by mistake. Your job is to calculate the angles that result in shooting yourself (so that you can avoid them).

The mirrors are not perfectly reflective, so we only need to worry about shots with at most 7 reflections. Below are several possible setups. In the figures the views are looking down from above. The mirrors are shown as black segments. All the paths that lead back to the firing point are shown in gray. Take the point of firing as the center of a Cartesian coordinate system, with postive x to the right and positive y up the page. The coordinates of the origin and the ends of the mirrors are shown. For simplicity assume the mirrors have negligible thickness. Each path is labeled with the initial firing angle, measured in degrees counterclockwise from the postive x axis and rounded to the nearest degree, 0 through 359.



Figures

Input: The input contains one or more data sets. Each data starts with a line containing the number of mirrors, n, $1 \le n \le 7$. The next n lines each contain the (x, y) coordinates of the ends of one mirror, so there is a sequence of 4 numbers for each mirror, $x_1\ y_1\ x_2\ y_2$. All coordinates are integers with magnitude less than 1000. No mirrors intersect or touch. No mirror passes through the origin.

After the last dataset is a line containing only 0.

Output: For each data set output a single line. If there are one or more paths back to the origin with no more than 7 reflections, collect the starting angles rounded to the nearest degree. All rounded angles, a, should be normalized so $0 \le a \le 359$. Eliminate any duplicates and print the integers out in increasing order on one line, using a single blank as separator. If there is no such path back, output "no danger". No path will hit the exact edge of a mirror. No line of output will have more than 79 characters. Please note that although the output is rounded to the nearest angle, your internal computations should be based on double-precision floating-point computations.

The first two example data sets correspond to the Figures, and the third merely omits the bottom mirror from the second data set.

Example Input:	**Example Output:**
2 2 1 -1 4 -2 4 -2 -1 4 3 10 13 10 -13 17 -7 23 33 17 27 23 10 -10 20 -10 3 3 10 13 10 -13 17 -7 23 33 17 27 23 0	45 67 90 135 157 180 29 61 63 117 no danger

Problem C: Pizza Pricing

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

Pizza has always been a staple on college campuses. After the downturn in the economy, it is more important than ever to get the best deal, namely the lowest cost per square inch. Consider, for example, the following menu for a store selling circular pizzas of varying diameter and price:

Menu	
Diameter	**Price**
5 inch	$2
10 inch	$6
12 inch	$8

One could actually compute the costs per square inch, which would be approximately 10.2¢, 7.6¢, and 7.1¢ respectively, so the 12-inch pizza is the best value. However, if the 10-inch had been sold for $5, it would have been the best value, at approximately 6.4¢ per square inch.

Your task is to analyze a menu and to report the *diameter* of the pizza that is the best value. Note that no two pizzas on a menu will have the same diameter or the same inherent cost per square inch.

Input: The input contains a series of one or more menus. Each menu starts with the number of options N, $1 \le N \le 10$, followed by N lines, each containing two integers respectively designating a pizza's diameter D (in inches) and price P (in dollars), with $1 \le D \le 36$ and $1 \le P \le 100$. The end of the input will be designated with a line containing the number 0.

Output: For each menu, print a line identifying the menu number and the diameter D of the pizza with the best value, using the format shown below.

Example input:	**Example output:**
3 5 2 10 6 12 8 3 5 2 10 5 12 8 4 1 1 24 33 13 11 6 11 0	Menu 1: 12 Menu 2: 10 Menu 3: 24

Problem D: Su-domino-ku

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

As if there were not already enough sudoku-like puzzles, the July 2009 issue of Games Magazine describes the following variant that combines facets of both sudoku and dominos. The puzzle is a form of a standard sudoku, in which there is a nine-by-nine grid that must be filled in using only digits 1 through 9. In a successful solution:

- Each row must contain each of the digits 1 through 9.
- Each column must contain each of the digits 1 through 9.
- Each of the indicated three-by-three squares must contain each of the digits 1 through 9.

For a su-domino-ku, nine arbitrary cells are initialized with the numbers 1 to 9. This leaves 72 remaining cells. Those must be filled by making use of the following set of 36 domino tiles. The tile set includes one domino for each possible pair of unique numbers from 1 to 9 (e.g., 1+2, 1+3, 1+4, 1+5, 1+6, 1+7, 1+8, 1+9, 2+3, 2+4, 2+5, ...). Note well that there are not separate 1+2 and 2+1 tiles in the set; the single such domino can be rotated to provide either orientation. Also, note that dominos may cross the boundary of the three-by-three squares (as does the 2+9 domino in our coming example).

To help you out, we will begin each puzzle by identifying the location of some of the dominos. For example, Figure 1 shows a sample puzzle in its initial state. Figure 2 shows the unique way to complete that puzzle.

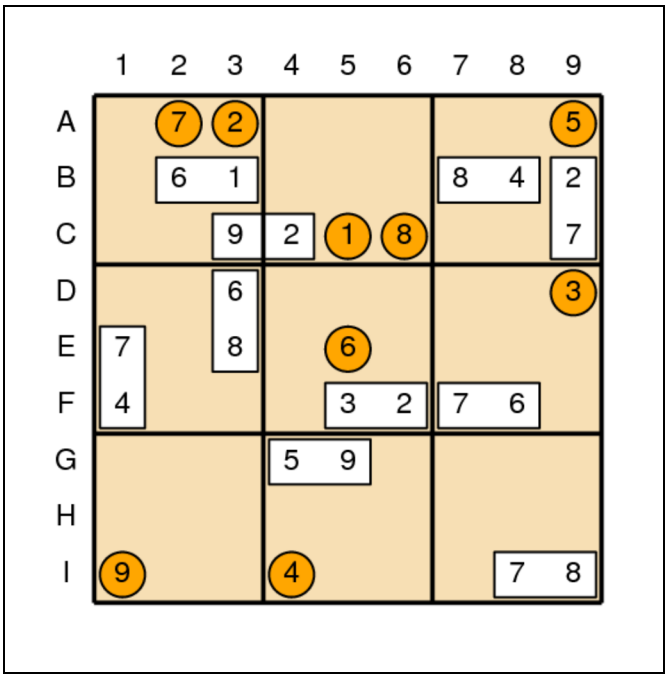


Figure 1: Sample puzzle

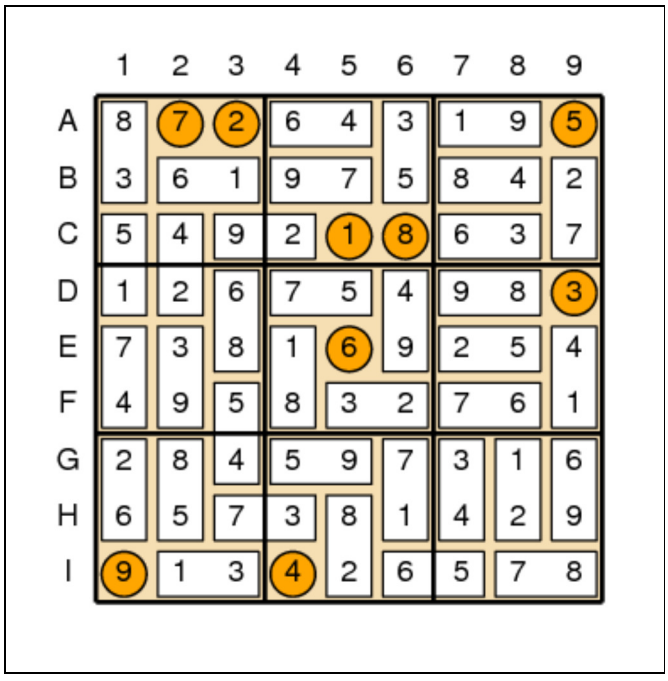


Figure 2: Solution

Input: Each puzzle description begins with a line containing an integer N, for $10 \leq N \leq 35$, representing the number of dominos that are initially placed in the starting configuration. Following that are N lines, each describing a single domino as $U\ LU\ V\ LV$. Value U is one of the numbers on the domino, and LU is a two-character string representing the location of value U on the board based on the grid system diagrammed in Figure 1. The variables V and LV representing the respective value and location of the other half of the domino. For example, our first sample input beings with a domino described as `6 B2 1 B3`. This corresponds to the domino with values 6+1 being placed on the board such that value `6` is in row `B`, column `2` and value `1` in row `B`, column `3`. The two locations for a given domino will always be neighboring.

After the specification of the `N` dominos will be a final line that describes the initial locations of the isolated numbers, ordered from 1 to 9, using the same row-column conventions for describing locations on the board. All initial numbers and dominos will be at unique locations.

The input file ends with a line containing 0.

Output: For each puzzle, output an initial line identifying the puzzle number, as shown below. Following that, output the 9x9 sudoku board that can be formed with the set of dominos. There will be a unique solution for each puzzle.

Example input:	**Example output:**
10 6 B2 1 B3 2 C4 9 C3 6 D3 8 E3 7 E1 4 F1 8 B7 4 B8 3 F5 2 F6 7 F7 6 F8 5 G4 9 G5 7 I8 8 I9 7 C9 2 B9 C5 A3 D9 I4 A9 E5 A2 C6 I1 11 5 I9 2 H9 6 A5 7 A6 4 B8 6 C8 3 B5 8 B4 3 C3 2 D3 9 D2 8 E2 3 G2 5 H2 1 A2 8 A1 1 H8 3 I8 8 I3 7 I4 4 I6 9 I7 I5 E6 D1 F2 B3 G9 H7 C9 E5 0	Puzzle 1 872643195 361975842 549218637 126754983 738169254 495832761 284597316 657381429 913426578 Puzzle 2 814267593 965831247 273945168 392176854 586492371 741358629 137529486 459683712 628714935

Problem E: Refrigerator Magnets

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

Like many families with small children, my family's refrigerator is adorned with a set of alphabet magnets: 26 separate magnets, each containing one letter of the alphabet. These magnets can be rearranged to create words and phrases. I feel it is my parental duty to use these magnets to create messages that are witty and insightful, yet at the same time caring and supportive. Unfortunately, I am somewhat hindered in this task by the fact that I can only make phrases that use each letter once.

For example, a nice inspiring message to leave for the children might be, "I LOVE YOU." Unfortunately, I cannot make this message using my magnets because it requires two letter "O"s. I can, however, make the message, "I LOVE MUSTARD." Admittedly this message isn't as meaningful, but it does manage to not use any letters more than once.

You are to write a program that will look at a list of possible phrases and report which phrases can be written using refrigerator magnets.

Input: The input will consist of one or more lines, ending with a line that contains only the word "END".

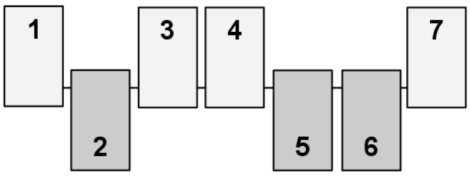
Each line will be 60 characters or less, and will consist of one or more words separated by a single space each, with words using only uppercase letters (A–Z). There will not be any leading or trailing whitespace, and there will not be any blank lines.

Output: Output only the lines which can be written in refrigerator magnets—that is, the lines which have no duplicate letters. Output them exactly the same as they were in the input—white spaces and all. Do not output the final "END" string.

Example input:	**Example output:**
I LOVE YOU I LOVE MUSTARD HAPPY BIRTHDAY GLAD U BORN SMILE IMAGINE WHATS UP DOC HAVE A NICE DAY END	I LOVE MUSTARD GLAD U BORN SMILE WHATS UP DOC

Problem F: Shut the Box

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



Shut the Box is a one-player game that begins with a set of N pieces labeled from 1 to N. All pieces are initially "unmarked" (in the picture at right, the unmarked pieces are those in an upward position). In the version we consider, a player is allowed up to T turns, with each turn defined by an independently chosen value V (typically determined by rolling one or more dice). During a turn, the player must designate a set of currently *unmarked* pieces whose numeric labels add precisely to V, and mark them. The game continues either until the player runs out of turns, or until a single turn when it becomes impossible to find a set of unmarked pieces summing to the designated value V (in which case it and all further turns are forfeited). The goal is to mark as many pieces as possible; marking all pieces is known as "shutting the box." Your goal is to determine the maximum number of pieces that can be marked by a fixed sequence of turns.

As an example, consider a game with 6 pieces and the following sequence of turns: 10, 3, 4, 2. The best outcome for that sequence is to mark a total of four pieces. This can be achieved by using the value 10 to mark the pieces 1+4+5, and then using the value of 3 to mark piece 3. At that point, the game would end as there is no way to precisely use the turn with value 4 (the final turn of value 2 must be forfeited as well). An alternate strategy for achieving the same number of marked pieces would be to use the value 10 to mark four pieces 1+2+3+4, with the game ending on the turn with value 3. But there does not exist any way to mark five or more pieces with that sequence.

Hint: avoid enormous arrays or lists, if possible.

Input: Each game begins with a line containing two integers, N, T where $1 \le N \le 22$ represents the number of pieces, and $1 \le T \le N$ represents the maximum number of turns that will be allowed. The following line contains T integers designating the sequence of turn values for the game; each such value V will satisify $1 \le V \le 22$. You must read that entire sequence from the input, even though a particular game might end on an unsuccessful turn prior to the end of the sequence. The data set ends with a line containing 0 0.

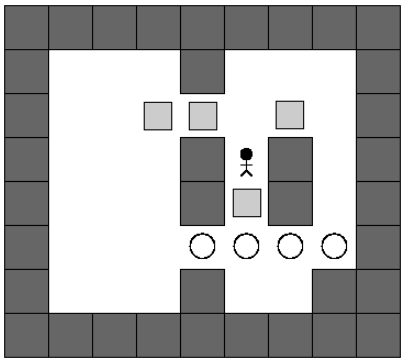
Output: You should output a single line for each game, as shown below, reporting the ordinal for the game and the maximum number of pieces that can be marked during that game.

Example input:	**Example output:**

`6 4`
`10 3 4 2`
`6 5`
`10 2 4 5 3`
`10 10`
`1 1 3 4 5 6 7 8 9 10`
`22 22`
`22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1`
`0 0`
	`Game 1: 4`
	`Game 2: 6`
	`Game 3: 1`
	`Game 4: 22`

Problem G: Sokoban

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



Soko-ban is a Japanese word for a warehouse worker, and the name of a classic computer game created in the 1980s. It is a one-player game with the following premise. A single worker is in an enclosed warehouse with one or more boxes. The goal is to move those boxes to a set of target locations, with the number of target locations equalling the number of boxes. The player indicates a direction of motion for the worker using the arrow keys (up, down, left, right), according to the following rules.

- If the indicated direction of motion for the worker leads to an empty location (i.e., one that does not have a box or wall), the worker advances by one step in that direction.
- If the indicated direction of motion would cause the worker to move into a box, and the location on the other side of the box is empty, then both the worker and the box move one spot in that direction (i.e., the worker pushes the box).
- If the indicated direction of motion for a move would cause the worker to move into a wall, or to move into a box that has another box or a wall on its opposite side, then no motion takes place for that keystroke.

The goal is to simultaneously have all boxes on the target locations. In that case, the player is successful (and as a formality, all further keystrokes will be ignored).

The game has been studied by computer scientists (in fact, one graduate student wrote his entire Ph.D. dissertation about the analysis of sokoban). Unfortunately, it turns out that finding a solution is very difficult in general, as it is both NP-hard and PSPACE-complete. Therefore, your goal will be a simpler task: simulating the progress of a game based upon a player's sequence of keystrokes. For the sake of input and output, we describe the state of a game using the following symbols:

Symbol	Meaning
.	empty space
#	wall
+	empty target location
b	box
B	box on a target location
w	worker
W	worker on a target location

For example, the initial configuration diagrammed at the beginning of this problem appears as the first input case below.

Input: Each game begins with a line containing integers R and C, where $4 \leq R \leq 15$ represents the number of rows, and $4 \leq C \leq 15$ represents the number of columns. Next will be R lines representing the R rows from top to bottom, with each line having precisely C characters, from left-to-right. Finally, there is a line containing at most 50 characters describing the player's sequence of keystrokes, using the symbols `U`, `D`, `L`, and `R` respectively for up, down, left, and right. You must read that entire sequence from the input, even though a particular game might end successfully prior to the end of the sequence. The data set ends with the line `0 0`.

We will guarantee that each game has precisely one worker, an equal number of boxes and locations, at least one initially misplaced box, and an outermost boundary consisting entirely of walls.

Output: For each game, you should first output a line identifying the game number, beginning at 1, and either the word `complete` or `incomplete`, designating whether or not the player successfully completed that game. Following that should be a representation of the final board configuration.

Example input:	**Example output:**
8 9 ######### #...#...# #..bb.b.# #...#w#.# #...#b#.# #...++++# #...#..## ######### ULRURDDDUULLDDD 6 7 ####### #..#### #.+.+.# #.bb#w# ##....# ####### DLLUDLULUURDRDDLUDRR 0 0	Game 1: incomplete ######### #...#...# #..bb...# #...#.#.# #...#.#.# #...+W+B# #...#b.## ######### Game 2: complete ####### #..#### #.B.B.# #.w.#.# ##....# #######

Problem H: Crash and Go(relians)

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

The Gorelians are a warlike race that travel the universe conquering new worlds as a form of recreation. Generally, their space battles are fairly one-sided, but occasionally even the Gorelians get the worst of an encounter. During one such losing battle, the Gorelians' space ship became so damaged that the Gorelians had to evacuate to the planet below. Because of the chaos (and because escape pods are not very accurate) the Gorelians were scattered across a large area of the planet (yet a small enough area that we can model the relevant planetary surface as planar, not spherical). Your job is to track their efforts to regroup. Fortunately, each escape pod was equipped with a locator that can tell the Gorelian his current coordinates on the planet, as well as with a radio that can be used to communicate with other Gorelians. Unfortunately, the range on the radios is fairly limited according to how much power one has.

When a Gorelian lands on the alien planet, the first thing he does is check the radio to see if he can communicate with any other Gorelians. If he can, then he arranges a meeting point with them, and then they converge on that point. Once together, they are able to combine the power sources from their radios, which gives them a larger radio range. They then repeat the process—see who they can reach, arrange a meeting point, combine their radios—until they finally cannot contact any more Gorelians.

Gorelian technology allows two-way communication as long as *at least one of them* has a radio with enough range to cover the distance between them. For example, suppose Alice has a radio with a range of 40 km, and Bob has a range of 30 km, but they are 45 km apart (Figure 1). Since neither has a radio with enough range to reach the other, they cannot talk. However, suppose they were only 35 km apart (Figure 2). Bob's radio still does not have enough range to reach Alice, but that does not matter—they can still talk because Alice's radio has enough range to reach Bob.

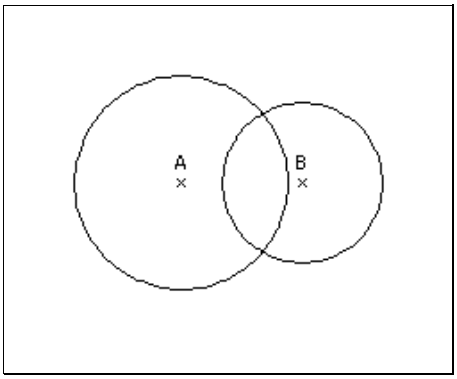


*Figure 1: Alice and Bob can **not** talk*

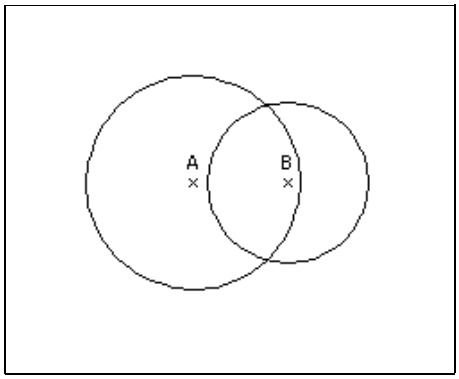


Figure 2: Alice and Bob can talk

If a Gorelian successfully contacts other Gorelians, they will meet at the point that is the average of all their locations. In the case of Alice and Bob, this would simply be the midpoint of A and B (Figure 3). Note that the Gorelians turn off their radios while traveling; they will not attempt to communicate with anyone else until they have all gathered at the meeting point. Once the Gorelians meet, they combine their radios to make a new radio with a larger range. In particular, the *area* covered by the new radio is equal to the sum of the *areas* covered by the old radio. In our example, Alice had a range of 40 km, so her radio covered an area of 1600π km. Bob's radio covered an area of 900π km. So when they combine their radios they can cover 2500π km—meaning they have a range of 50 km. At this point they will try again to contact other Gorelians.

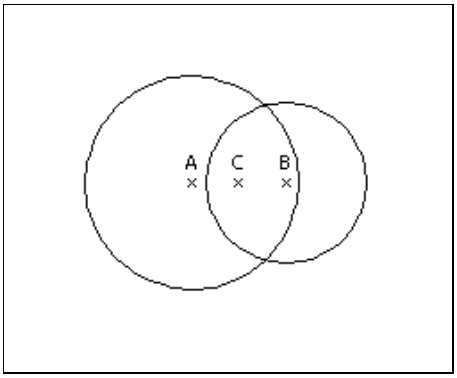


Figure 3: Alice and Bob agree to meet at the midpoint

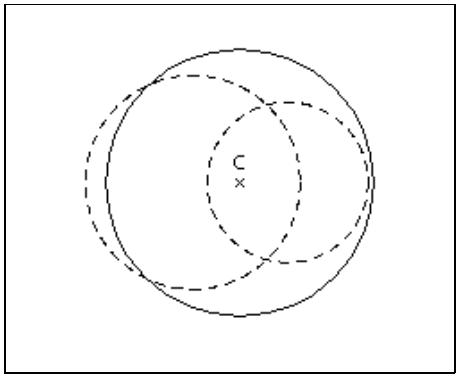


Figure 4: Alice and Bob combine their radios

This process continues until no more Gorelians can be contacted. As an example, suppose the following Gorelians have all landed and all have a radio range of 30 km: Alice (100, 100), Bob (130, 80), Cathy (80, 60), and Dave (120, 150). At this point, none of the Gorelians can contact anyone else (Figure 5). Now Eddy lands at position (90, 80) (Figure 6). Eddy can contact Alice and Cathy, so they arrange to meet at (90, 80), which is the average of their locations. Combining their radios gives them a range of $\sqrt{2700} \approx 51.96$ km (Figure 7).

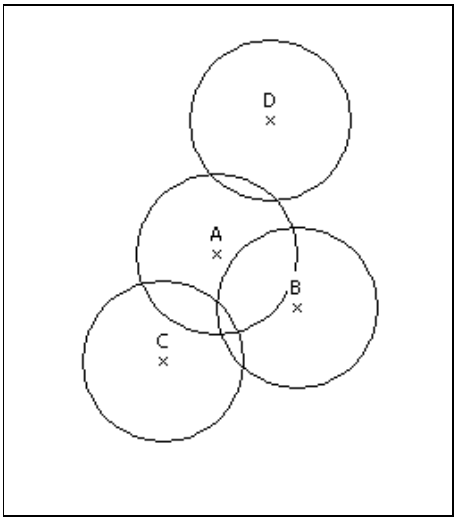


Figure 5: Nobody can talk

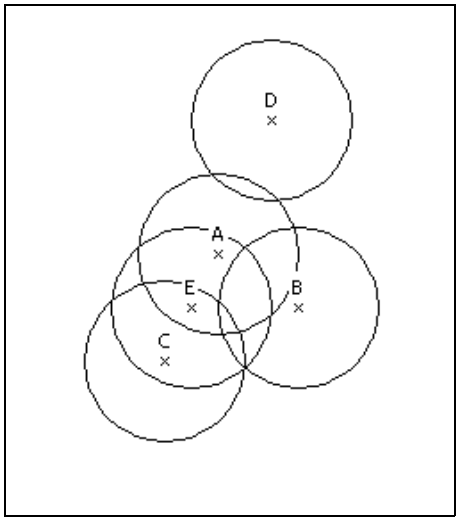


Figure 6: Eddy joins the group

Now they check again with their new improved range and find that they can reach Bob. So they meet Bob at (110, 80) and combine their radios to get a new radio with a range of 60 (Figure 8). Unfortunately, this is not far enough to be able to reach Dave, so Dave remains isolated.

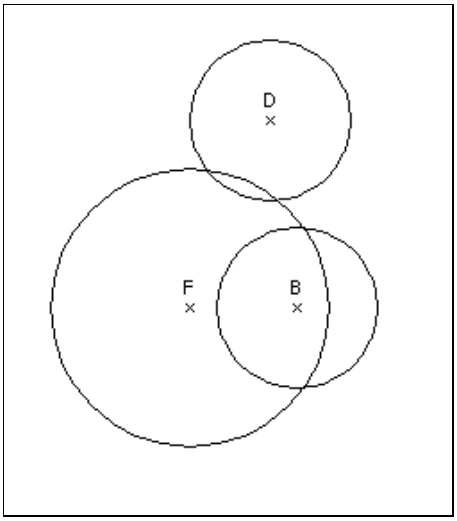


Figure 7: Alice, Cathy, and Eddy team up

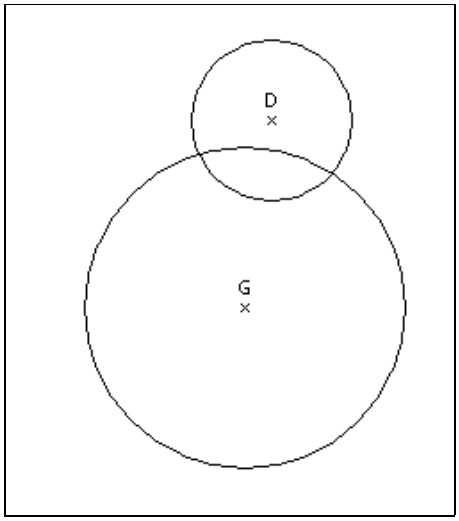


Figure 8: Bob joins the group

Input: The input will consist of one or more data sets. Each data set will begin with an integer N representing the number of Gorelians for this dataset ($1 \le N \le 100$). A value of $N = 0$ will signify the end of the input.

Next will come N lines each containing three integers X, Y, and R representing the x- and y-coordinate where the Gorelian lands and the range of the radio ($0 \le X \le 1000$, $0 \le Y \le 1000$, and $1 \le R \le 1000$). Note that only the Gorelians' *initial* coordinates/range will be integral; after merging with other Gorelians they may no longer be integral. *You should use double-precision arithmetic for all computations.*

The Gorelians land in the order in which they appear in the input file. When a Gorelian lands, he merges with any Gorelians he can contact, and the process keeps repeating until no further merges can be made. The next Gorelian does not land until all previous merges have been completed.

Output: The output will be one line per data set, reporting the number of independent groups of Gorelians that remain at the end of the process.

Example input:	**Example output:**
5 100 100 30 130 80 30 80 60 30 120 150 30 90 80 30 6 100 100 50 145 125 10 60 140 15 160 145 20 130 135 25 80 80 30 0	2 3