Twenty-Ninth Annual
Willamette University—Oregon Computer Science Teachers' Association High School Programming Contest

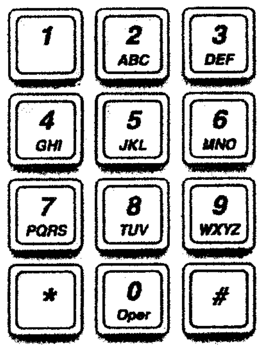
Saturday 14 March 2015

Contest Problems

(In the problem statements we assume that input/output will be done "console style" and we try to describe input formatting issues such as separation of parts by blank lines, etc. You are welcome to **prompt** *for the input as you need it, or even to write a simple* **GUI** *, but you should accept the basic data as described.)*

1. Penelope's Phone Peril

In a story right out of TV drama, an undercover special agent is in a bad spot: he's trapped in the bad guys' lair and needs to communicate with his team on the outside, but he can't risk blowing his cover. The bad guys in question run an evil ... *telemarketing scam!* The only way he can contact his team is through the phone he's using to make calls while in his undercover identity. By advance arrangement, he'll sometimes call non-existent phone numbers—these will be noted and routed to his team by their quirky tech expert, Penelope. Our special agent will send a few pre-arranged messages in the "wrong numbers" he calls, using the *alphabetic correspondence* from old-style phone keypads to embed a message in the *phone number itself.*



EXTRACT	=	**3987228**
HELP	=	**4357**
SOLO	=	**7656**
PIZZA	=	**74992**

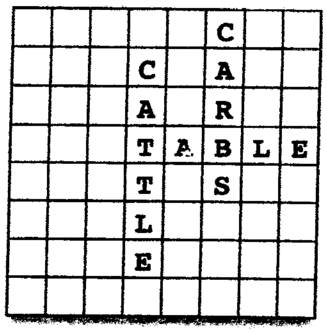
Your input will consist of two parts: the first will be a series of keywords that make up the "dictionary" of pre-arranged messages, things like "`EXTRACT`", "`HELP`", "`SOLO`" (meaning he's alone), "`PIZZA`" (meaning he's hungry and a delivery should be made), etc. The keywords will be given in all uppercase and separated by spaces—they may be on separate lines, but will always be terminated by a blank line (or you may use a GUI if you wish). The second part of the input will be a series of 10-digit numbers, one per line, with no spacing or punctuation (we recommend you read them as strings). Each number will include *zero or more* of the previous code words, *encoded according to the phone pad.* Code words will be no more than 10 letters and will be embedded as *consecutive digits* in the phone number.

In your output, you should report for each 10-digit number *all the coded messages* it contains, or explicitly state that there *were* no coded messages, if no embedded ones were found.

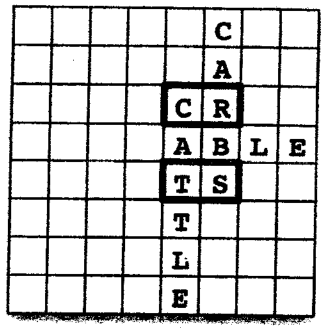
2. Letter *scramble!*

Many popular word games involve placing "letter tiles" on the squares of a grid in order to form words. You've just been hired by a company making a new smart-phone version of one of these word games. Your job is to check and see whether a given set of words can be put on the board in a single "play"—other members of your dev team will be responsible for generating candidate words and for checking whether a play interferes with words already on the board. In other words (heh), you need to check *only* for the placement of the words you get as input, on a blank board area of a given size. This will make the problem easier ... but you also have to place multiple words at a one time, unlike in some real games, so that will be rather a bit harder!

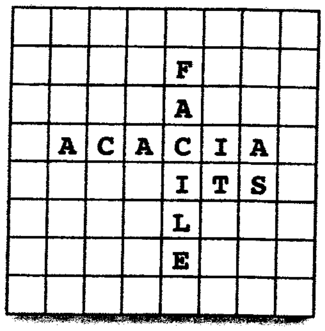
Your input will include first a number (the size of the square board) and then a sequence of words, all made from uppercase letters, separated by spaces on a single line (or entered into a GUI text field, if you like). You should find a way to place *all* the words on a board of that size, but with these conditions: (1) every word must connect or overlap with at least one other, *joined by some common letter,* so that no word is disconnected from the others; (2) all the words must be used together in a single play (unlike some real word-tile games); (3) each word must be laid out *horizontally (left-to-right)* or *vertically (top-to-bottom);* and (4) any two *adjacent letters* must be part (or all) of some word given to you.



valid layout for 3 words 'CATTLE', 'CARBS', 'TABLE'



NOT valid for 3 words, because of 'CR' and 'TS'



valid layout for 5 words including 'IT', 'ITS' and 'AS'

In the example boards above, the first board is a valid placement for the words CATTLE, TABLE, and CARBS. The second board shows an ***invalid*** play for the same words, since it implies that the "words" CR and TS were part of the input. On the other hand, the last board is perfectly valid, just as long as the input includes all the words: ACACIA, FACILE, IT, AS, and ITS. Note in particular that IT and AS are forced as words due to the placement of ITS below ACACIA. Finally, note that when letters are used to connect a word, that "uses up" all the letters of both words—unlike in real tile games, we don't have separate tiles (if you like the tile metaphor, you could imagine them being stacked on top of each other at the intersections).

Let's number the board from position (1,1) in the lower left, running to (n,n) in the upper-right, where n is the board size. (Thus horizontal words have an increasing x coordinate as they run from left to right, but vertical words have a *decreasing* y coordinate as they run from top to bottom.) Your output should consist of either (1) a report that the words *cannot* be placed, *even if it is just because some word is too long,* or (2) a listing with each word, the coordinates of its first letter, and whether it runs horizontally or vertically. For example, an output for the arrangement in the leftmost board above could be:

```
CATTLE at (4,7) vertical;  CARS at (6,8) vertical;  TABLE at (4,5) horizontal
```

3. Fraction compaction

Many computer programs use *floating-point numbers* to do calculations involving fractions—those numbers are stored in binary (base 2) on modern computers, and normally use the *IEEE 754 Standard for Floating-Point Arithmetic.* Unfortunately, even a good and thorough standard like the IEEE 754 can't make up for the fact that some simple fractions can't be represented in binary without some loss of precision. For example, you can represent halves and quarters exactly in base 2, but not thirds! So, when fractional numbers are stored this way, and calculations are performed on them, rounding errors and other issues inevitably occur.

There *is* a way around this, however: if you want to work with fractions and do *exact* computations, you can keep track of the numerators and denominators separately, and use the usual middle-school rules to perform operations like addition, subtraction, multiplication and division. For example, adding two numbers involves re-working the fractions to have a common denominator and then adding the numerators. Multiplying is a bit easier: just multiply both the numerators and the denominators separately and re-combine. Subtraction and division are similar—you should know the rules from school. In any case, just as in school, we should also simplify to *lowest common terms,* either as a final step or even along the way. (Fractions are in "lowest common terms" when the numerator and denominator have no common whole number multiple—for example, 2/3 is in lowest terms, but 4/6 is not).

So what's the problem, then? We will input to your program a line of text with some fractions on it (always *positive* whole numbers in both parts, separated by a slash '/'). Between each pair of numbers will be an *arithmetic operator,* one of +, *, –, or / (careful with that last one, the 'slash', as it's the same character used between the parts of a fraction!). There will always be spaces between fractions and operators, but never between the parts of a single fraction. You should calculate *a cumulative fraction result* by starting with the first fraction and performing the given operations on a "running total" as you go. ***Notice that there are no parentheses, but we are NOT using the normal "order of operations"!*** Instead (and this makes it easier), you should just perform the operations in the order given, from left to right.

There will never be more than 10 fractions on the input line (and so never more than 9 operations). Our input fractions will always be positive, *but they may not be in lowest common terms (sorry!).* You should report the final result of all the operations as a fraction *in lowest common terms,* using the same 'slash' syntax. If a fraction is negative, put a negative sign to its left—this won't happen in the input, but it may be necessary in your output. (Just remember to work out the proper sign for the fraction *as a whole,* based on the signs of numerator and denominator.)

```
> 2/3   +   4/5   *   1/2   -   1/4

29/60

> 12/3   *   7/8   -   133/11   *   4/6

-63/11
```

4. Goldilocks and the binary digits

Take an integer n and convert it into base two (in other words, what is called its *binary form,* as 1s and 0s, but with *no leading 0s*). Now compare the number of 1s to the number of 0s: imagine that the 1s are heavy, like water, and the 0s are light, like bubbles of carbonation. If the binary representation has more 1s than 0s, it is "`Too heavy`"; if it has more 0s than 1s, it is "`Too light`"; and if it has the same number of each, it is "`Just right`".

Your program should read in positive integers (between one and one billion, inclusive) and report whether they are too heavy, too light or just right. For example, on input `292` your program should produce the output "`Too light`" since the binary representation of `292 is 100100100` (remember, no leading 0s.)

A Special Note About Pi Day!

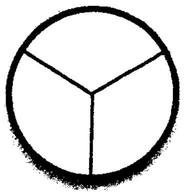
Today's date, 3-14-15, is being celebrated around the world as "Pi Day" because it looks like the beginning of the decimal expansion of the real number $\pi = 3.1415926\ldots$. You can read more about Pi Day at the website `<www.piday.org>` ... once the contest is over, of course (no internet access during the contest, please)! So, in honor of Pi Day, the next two problems will involve ... pies!

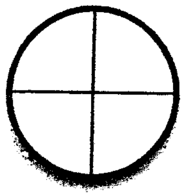
---

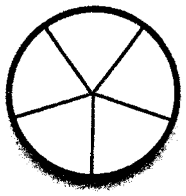
5. Pie-cutting conundrum

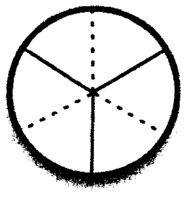
You work at a northwest regional restaurant (called Cheri's) that specializes in baking and serving pies. Sometimes customers arrive in large groups and order a lot of pie slices, especially on Wednesdays, when the pie comes free with dinner. Depending on your stock of pies, it can be difficult to serve everyone in a large group with a standard-sized slice. But management would rather at least give a *smaller* slice to *everyone* than to run out and leave some customers pie-less. Since you serve so many pies, you have a standard set of "quick-cutter" tools to cut pies into multiple (equal) slices in one step. These are like squat, hollow cylinders with dull blades radiating from the center: place one over a pie and push down, and you slice the whole pie into a specific number of equal slices. If you turn the cutter just the right amount, you can cut the slices into halves again, thus getting twice the number of slices (but still equal-sized, if you're careful).

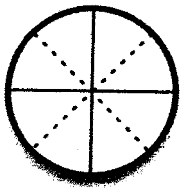
As it turns out, you have quick-cutters with 3, 4, and 5 blades, so you can easily cut any pie into 3, 4, or 5 slices; with careful turning, you can also cut pies into 6, 8, or 10 slices; or even 9, 12, or 15 ... and so on, for any multiple of 3, 4, or 5 (a 1/20th slice of pie is pretty sad, but it's better than nothing!). Unfortunately, there's no way you can easily cut a pie into 7, 11, 14, or 22 equal slices, as the cutter trick won't work for these numbers. So, here's the problem: when a group of people comes in, and you have a certain number of pies ready to go, how should you slice those pies so that: (a) everyone gets an equal sized slice; (b) the slices can be cut with the available quick-cutters; and (c) the slices are as large as possible. In particular, you are only allowed to use *just one* of the three quick-cutters, turning it if necessary, but always the same amount for each pie. Once you're done cutting, you serve exactly one slice to each customer.

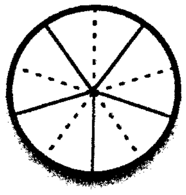

3 slices


4 slices


5 slices


6 slices


8 slices


10 slices (!)

The input will be two numbers: the number of customers that come in, and the number of pies available. Both numbers will be positive, and the number of customers will always be at least 3 *times* the number of pies—that way the use of the cutters is always justified (a whole or half pie would be too big anyway!). Your output should consist of a message describing *which sized cutter to use,* and *how many slices per pie.*

```
> 40  7                          (that's 40 customers and 7 pies)

Cut 7 pies with the 3-way cutter, into 6 slices each.

> 27  2

Cut 2 pies with the 5-way cutter, into 15 slices each.
```

6. Pie-packing puzzler

Well, you're on the job at the Cheri's pie restaurant again, but this time you have a new problem: you are shipping pies from your in-house bakery to some of your outlying stores and you need to determine the full weight of the pie shipment *before* you put it on the truck. (Last week your co-worker's plan was: "Just load 'em up and see how many will fit" — for which he got a broken truck axle and some serious dinner delays!)

As it turns out, pies of different flavors have different *densities* (think: cherry versus chocolate cream) and also different *heights* (think of that lofty lemon meringue!). So, in order to determine the weight* of all those pies, you have to multiply their volume by their density—luckily the *radius* is the same for all the flavors, exactly 5 inches, since they all have to fit in the same baking pans. You should be familiar with the formula to determine the area of a circle, or of a circular pie: that's just π times the radius squared, or πr^2 ... even though these pies are round (say the last formula out loud if you don't get the joke). Then height times area equals volume! Your pie trucks have a fixed number of pie racks, all holding the same number of pies, and the trucks are always filled with one flavor per rack. The number of racks depends on the truck—what you need to calculate is the total pie weight.

Your input will consist of one line with 2 positive integers, followed by several lines with 2 floating-point numbers each (that is, numbers with decimal points). The first integer is the number of pies per rack in this truck, the second is the number of racks, and thus the number of flavors. The remaining lines, one per flavor, provide the density (in pounds per cubic inch) and height (in inches) of each flavor. For your output, you should report on the total weight* of the pies for this truck, in pounds.

```
> 12    3              (that's 12 pies per rack, and 3 racks/flavors, which follow in the input)
  0.4   2.0
  0.2   3.5            (looks like lemon meringue to me: light but tall)
  1.1   1.5

Total weight =   2968.812 pounds
```

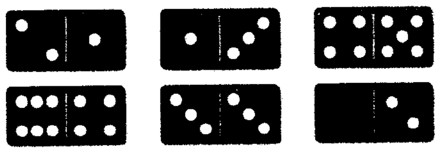
* *(Yes, I know: weight is different than mass; but our "densities" are expressed in terms of pounds, just for convenience.)*

7. Piling on the dominoes

Remember those little game tiles you played with as a kid? Dominoes! Each domino is just a flat, black rectangle, divided into two halves by a line: let's agree to call the two halves *tiles*. Each domino tile contains from zero to six white dots, grouped in patterns like those on dice. People play all kinds of games with dominoes, often placing them on a floor or table according to some rules about how the numbers on adjacent dominoes match. We are going to place our dominoes on a square grid, but one of limited size. Since we may have more dominoes than can fit, we will pile them up on top of each other, if necessary, in order to place them all. Our rules for adjacent sides of the dominoes is different than the usual: if the tiles of two dominoes are adjacent (or touching), one tile should have an *even* number of dots and the other an *odd* number of dots ... except for zero dots, which acts as a "wild card", matching anything it's adjacent to.

In fact, we will insist on this rule *even if the way the tiles touch is by being on top of one another.* Of course, two tiles on the same domino may not obey the even/odd rule, but that's not *our* fault: it's just the way the domino comes to us. One more rule about piling up dominoes: we insist that every domino be placed so that each of its tiles is supported from below, either by the "ground" surface, or by another domino. So no free-floating dominoes, and no tiles hanging off into space—after all, this is not Jenga!

An example will probably help at this point: here is a set of dominoes along with the input format we will use to describe them (each domino has a "decimal point" separating the numbers on its two tiles):

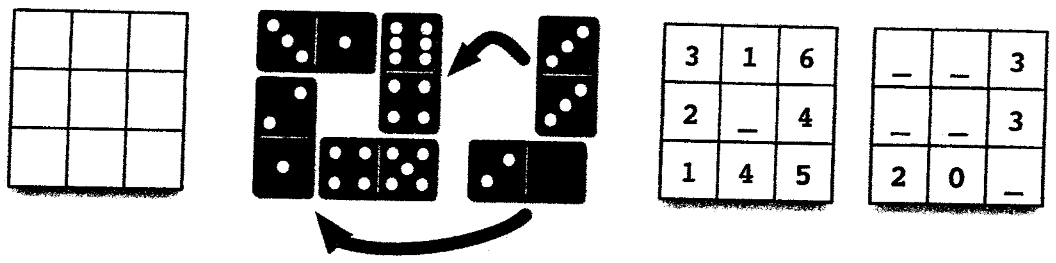


```
2.1   1.3   4.5

6.4   3.3   0.2
```

On the next page we show a 3-by-3 grid and an arrangement of the dominoes above so that they fit on the grid, obeying all the rules. The arrows show where a couple of the dominoes will be placed on the "next level up". The two grids on the right show how we can describe the final arrangement: we use one grid per level (starting with the bottom), and write the number of dots for the corresponding tile in the grid position; if no domino tile sits in a position, we just write an underscore. (For actual output you don't need to draw the grid, just write out numbers and underscores in an arrangement matching the square grid size.)

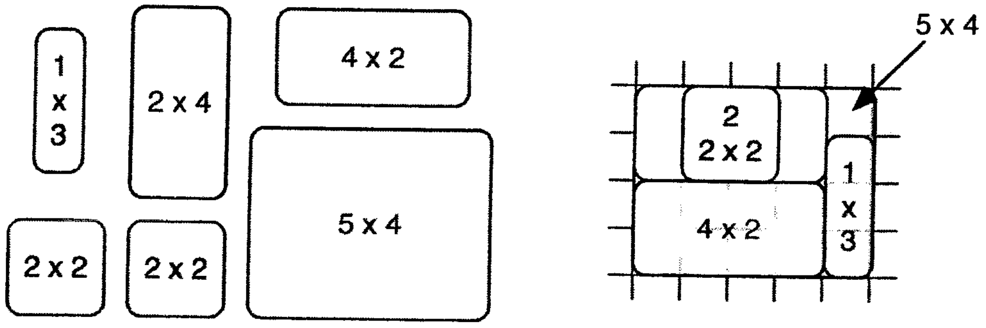
So, your input will consist of one number (say on a separate line) representing the square grid size, and then a series of "dotted pairs" of numbers, representing the dominoes (we will put the dominoes on several lines, if necessary, and terminate with a blank line.) Your output should consist of one or more grid-shaped arrangements of numbers-and-underscores, one per level from, the bottom up, as in the sample picture. *If you cannot arrange the dominoes in the given grid according to the rules, print a message to that effect!*



(Note that you are not required to fill every grid space, even on the lowest level—this means that you might use a 2-by-2 arrangement, or even smaller, within the confines of a 3-by-3 or larger grid; you'll just need to pile higher!)

8. Nested storage

Everyone has too much stuff these days: just take a look in your local discount store, where clear plastic storage boxes to hold all that stuff are selling like hotcakes! As a clerk at the local store, you pack up a lot of storage box orders: it's a bit of a chore, but you try to make it fun by seeing if you can pack a whole box order together inside the biggest box. One box may be nested inside another if the first is smaller *or even the same size!* Furthermore, several smaller boxes may be placed next to each other inside a bigger box if they fit into the space available (to make things simple, we'll assume that the boxes are always aligned either east-west or north-south: no diagonals, please!). Finally, boxes may be nested to any depth. Here is an example of how some boxes might be packed together—the original boxes are on the left, and they are nested and packed on the right:



Note that you may need to turn a box 90 degrees in order to get it to fit well. In the example, the two 2-by-2 boxes have been nested inside each other, then nested inside the 2-by-4 (which was rotated to fit), which is in turn nested inside the 5-by-4, next to the 4-by-2 and the 1-by-3 (the 2-by-4 is in gray to make it more visible).

Your program should read in a series of numbers, interpreted as pairs of lengths and widths, terminated by a couple of zeros if you need it (the zeros don't represent an actual box). Your program should determine if there is some way to pack the all the boxes together inside the largest box according to the rules above. The output should list the boxes using their dimensions to identify them (possibly switched, if the box got rotated), and give their absolute `(x,y)` positions on a grid, viewed from overhead (as in the example above). The coordinates represent where the lower-left-hand corner of the box is placed, with x coordinates increasing to the right and y coordinates increasing up the page. Just to be definite, let's agree to place the bottom-most box with its identified corner at `(0,0)`.

Finally, when one box is nested inside another, it should be listed *after* the box it is nested inside, so that boxes are listed from the bottom up to the top—this order would be convenient if you were reading the list as a series of instructions on how to place the boxes, putting down the bottom box first, then some others on top of it, etc. Adjacent boxes can be listed in any order, but you should finish a single "stack" of nested boxes before going back down to fill in any adjacent ones nearer the bottom.

For example, here is one way of listing out the box arrangement above:

```
5 by 4   at   (0,0)
4 by 2   at   (0,0)
4 by 2   at   (0,2)
2 by 2   at   (1,2)
2 by 2   at   (1,2)
1 by 3   at   (4,0)
```

If the boxes don't fit at all, print a message saying that. But if they do fit and you print a list, every box must be included in the list, either as originally dimensioned or else rotated. Finally, there's no need to search for the "best" fit: any way of packing them all together will do. If, however, the boxes *cannot* be stacked up together, all inside the "biggest box", your program should say so. (How can this happen? Just imagine a longer, skinnier box and a shorter, wider one: neither will fit inside the other!)

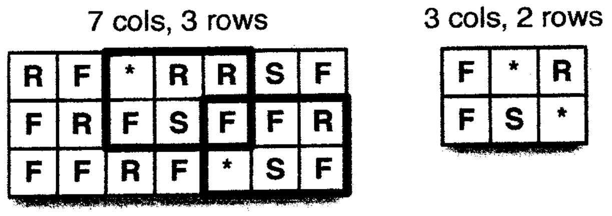
9. Terrain-spotting

The Oregon Department of the Environment has a job for you: they are analyzing satellite photos of various wilderness areas, looking for certain patterns of land features. Each satellite photo is a grid of features, coded as letters, e.g., **F** for forest, **S** for swamp, **R** for river, etc. The patterns they are looking for are a kind of "template", also expressed as a grid (but a *smaller* one), and using the same feature codes. Furthermore, both grids may have asterisks (*) in them: for the satellite photos, this means there was a glitch in the sensor and the exact feature type is unknown; for the pattern grids it means "we don't care what feature is here". In any case, an asterisk in either grid matches *any* feature in the other grid (for sensor glitches, the department wants to err on the side of a possible match: a specialist will be sent into the field for a better look).

Your program will be given a satellite photo grid and a pattern grid: it should report *all* the places in the satellite photo that match the pattern, i.e., every (column, row) position such that, if the **upper-left** corner of the pattern is overlaid on the satellite photo at that position, all the features in the pattern grid will match the corresponding features in the satellite grid underneath. (Remember, asterisks in either grid match *any* feature in the other grid—including asterisks!)

Each grid will be given to you as a pair of integer numbers (the total numbers of columns and rows, in that order) followed by a string of uppercase letters (and asterisks) of appropriate length (columns times rows). The features are meant to fill each row from *left-to-right first,* and the rows *from top to bottom.* (Note: this is a different ordering than some of our other problems, but it seems more natural in this context.) *Columns and rows will be numbered starting from 1 for the purposes of reporting a match.*

For example, the following satellite and pattern grids would match in the bold, outlined positions:



For this example the input and output would look like this:

```
Photo size? 7 3
Features?    RF*RRSFFRFSFFRFFRF*SF

Pattern size? 3 2
Features?    F*RFS*
```

```
Matches at:
   position 3 1
   position 5 2
```

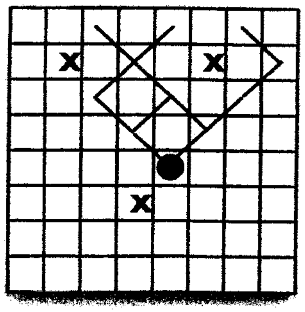
10. Checker challenge

A checkerboard has eight rows and columns of squares. Let's number the squares so that the lower left one is square (1,1) and the upper right is square (8,8). Checkers can move diagonally forward, to the left or to the right, and they are not allowed to move onto a square that is already occupied by another checker. (Our checkers will not do any jumps, just plain moves.) So a checker starting at some point on the board will have several *paths* it can take to reach the other side of the board (i.e., row 8): no path is allowed to go through an occupied square, but there might still be a lot of options available. We can represent each path as a sequence of moves, "`L`" or "`R`", for diagonally forward to the left or right, on each turn.

Your program should read in the "coordinates" of a starting square and some occupied squares of a board, and should print out a list of all "paths" that can be taken by a checker. *You should report if there is no path or if the piece is already on the last row.* The input will consist of one line containing the coordinates of the starting square, then several lines each with the coordinates (column, then row) of the occupied squares, as two integers. If you need it, you may request a line with `0 0` to indicate the end of the data.

Sample input (in the picture, **x** is used to mark an occupied square, and our checker is the black dot):

```
5  4
4  3
2  7
6  7
0  0
```

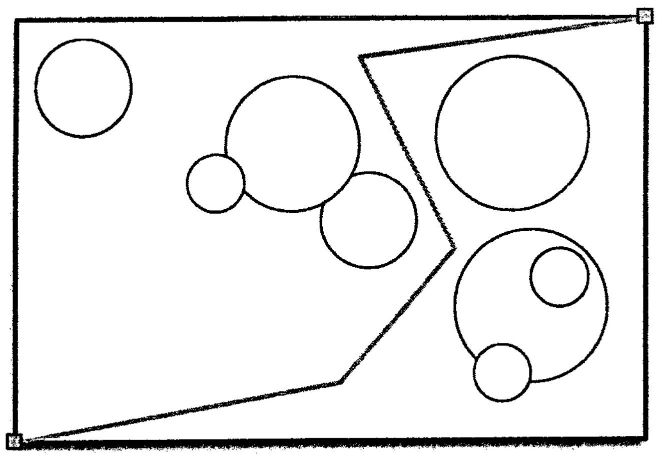


There are seven possible paths to the last row, so your program should show:

```
LLRL
LLRR
LRLL
LRLR
RLLL
RLLR
RRRL
```

11. Zombie zig-zag

Oh noes, it's the Zombie Apocalypse™! You find yourself caught out on a mission away from the camp where your rag-tag bunch of hold-outs desperately clings to hope and safety. Now that night has fallen, the zombies will begin to move again! You have a map which shows the known positions and movement speeds of all the zombies spread across the (rectangular) field between you and your camp (zombies move at varying rates of speed due to missing limbs and other grisly disfigurations). You're considering making a 5-minute dash across the field and have worked out a "danger zone" for each zombie. The zone is a circular area, centered on the zombie's original daytime position, and with a radius that corresponds to its maximum range, based on its speed, during your 5-minute sprint. In order to play it safe, you won't assume anything about the zombies' direction of movement, so you want to plan a path that will avoid all the circular danger zones. You'll start at the southwest corner of the field, and try to make it to the northeast corner without ever entering (or even touching) any zombie's circle. (See the map on the next page for an example.) Sometimes the zombies may be so thick and fast that you won't be able to make it safely across the field—but if you can, we want your program to provide a safe path, described as a series of points, starting at the southwest corner (with coordinates (0,0)), and ending up at the northeast corner, such that none of the line segments defined by your path intersects a zombie "danger zone" circle.



As input, we will provide the overall dimensions of the field (2 floating-point numbers, representing the (x,y) coordinates of the northwest corner), followed by a number of lines, each with three floating-point numbers. The numbers on each line represent a circle, where the first two are the (x,y) coordinates of the center and the third is its radius.

As output, you should either provide a message of despair telling that no path is possible, or you should provide a path as a series of points, as (x,y) coordinates, running from corner to corner and avoiding all circles. (Note that you should also stay within the confines of the rectangular field at all times.)

12. Operator fixer-upper

Your pal Russell is a fan of the LISP family of programming languages (these include LISP, Common LISP, Scheme and Racket, just to mention a few). One of the characteristic features of these languages is that they use a *prefix* syntax, even for familiar arithmetic operators and expressions—most other languages use an *infix* style, as we do in everyday mathematics. For example, a LISP programmer would write `(+ 2 (* 3 5))` to represent the arithmetic expression most of us would write as `(2+(3*5))` (well, most people wouldn't use an asterisk for multiplication, but we programmers do!). Note that the left and right arguments of an operation come in the same order in both notations (this matters for subtraction and division), and that we will use "full parentheses" for both input and output.

You and Russell are collaborating on a project and you need to put a bunch of formulas he worked out into a program you're writing. Unfortunately, Russell's contributions are all written in this odd (to you) prefix style—you can't put them directly into your program, which is written in a language with "normal" infix syntax. Worse yet, Russell has given you quite a bit of material, and you don't have time to convert it by hand—even if you did, you're worried that you would introduce lots of errors along the way.

The obvious strategy is to write a *program* which will convert Russell's prefix-style expressions into the infix form, so that you can then add them to your code. Fortunately, the math involved is pretty basic, just the four standard operations (+, *, – and /), non-negative integers, and some variable names (each variable will be just a single lowercase letter). Russell's inputs always have spaces between operators, numbers, and variables, but not necessarily next to parentheses. For your output, you might as well use full parentheses, too: it will be easier to produce, since you won't have to figure when to leave them out, in favor of "order of operations". And, anyway, the computation will work the same, even if there are redundant parentheses in there. For example, here is an input from Russell and your corresponding output:

```
> (+ (* x 3) (- (* y 5) (/ x 7)))

  ((x * 3) + ((y * 5) - (x / 7)))
```

13. Spaceball!

Sometime in the not too-distant future, in a facility in low-earth orbit and thus not *too* far away ... a couple of astronauts are floating in a zero-gravity game room—basically a large "3D rectangle" or hollow box. They are playing a variation on the earthly game of handball called *SpaceBall,* which involves bouncing a small rubber ball off the walls of the room. Each astronaut is floating motionless at some position in the room, designated by (*x,y,z*) coordinates (in particular, they won't move from that position during a single play of the ball). One of the astronauts will throw the ball, aiming at some point on one of the walls: the ball will fly in a straight line, hitting that point on the wall (our keen-eyed astronauts never miss!), and then bouncing off *according to the law of equal-but-opposite angles* (but in 3D!). The ball will then fly in another straight line, eventually hitting a new point on another wall, then bounce again, and so on. (Computing the bounce points, angles, and lines-of-flight will be one of your program's tasks.)

The goal of the game is to throw the ball in such a way that it will eventually fly toward the other astronaut: the thrower's opponent must try to catch the ball if it comes within his reach, and if not, he loses a point when it hits him or passes nearby. How near? Well, we will model the astronaut's reach as a sphere around his position point (you might think of Da Vinci's famous drawing of a figure's limbs fitting roughly inside a circle). We won't worry about whether the opponent actually catches the ball or not, just whether it enters their "sphere of influence". However, the rules of the game also allow for the possibility that the thrower will end up bouncing the ball back in their *own* direction. So your job will be to determine whether the ball will enter the sphere of either astronaut, the original thrower or their opponent, and to tell which. In order to put a reasonable limit on things, we'll say that the game rules allow for the ball to bounce a *maximum of four times off the walls,* including the initial bounce, before it goes out of play (assuming it doesn't "hit" one of the players first).

OK, that's the basic set-up: what about input/output details? We will provide your program with all of the following:

(1) the 3-dimensional size of the room, expressed as three floating-point numbers (say on a line);

(2) for each of the two astronauts, a line with a position and a radius of reach, thus defining a sphere: each line will have four floating-point numbers, three for coordinates of the center and one for the radius (so that's two lines each of four numbers ... or use a well-labelled GUI, if you prefer);

(3) the position of the first bounce point, which will always be on one of the six walls of the room; we'll express this as a 3D point, i.e., with three floating-point numbers. (For simplicity, we'll assume that the first line segment on the ball's path starts from center of the first astronaut's sphere.)

You should compute the path of the ball in 3D space as described above, then provide the following output:

(1) a sequence of 3D points (say one per line) representing the bounce points on the walls—you should echo the *first bounce point* (which we gave you) just for clarity;

(2) there should either be four bounce points in this sequence (at which point you may stop and declare the ball out of play) or you should stop early and tell which astronaut's sphere was intersected on the way from the last-mentioned bounce point. (You may refer to them as "first" and "second", or as "thrower" and "opponent".)

Hint: in order to understand the way the ball bounces, you might think of projecting its path against one or more of the walls: it is easier to see how the paths go in 2D than in 3D. On the other hand, you still have to come up with a way of computing the path based on this (conceptual) understanding.

14. Text expander

Your poor grandparents are doing their best to "get with it" and join the texting revolution! They have smartphones now, but they just can't get the hang of all this new-fangled jargon (as grandpa calls it): things like "`lol`", "`brb`", "`rofl`" ... even simple "smileys" or emoticons, like "`:)`" or "`<3`", throw them off completely. ("Why does your sister keep saying 'less than three'? Less than three what?" complains grandma.)

Well, lucky for them, they have a programmer in the family! Your program will first read in a set of abbreviations like those above, along with an "expanded form" for each one. It will then read in a longer text message which may contain uses of the abbreviations. Your program will expand every abbreviation "in place" into its expanded form. However, because this is taking place in a narrow chat window on a smartphone, you will also have to wrap the output to a given fixed width, and left-justify it therein.

Your input will therefore start with a single positive integer, representing the fixed width of the chat window. This will be followed by several lines of text: each will start right off with a single string, possibly containing punctuation or the like, but with no blanks in it (this is an abbreviation). On the same line, after a few blank spaces, will come the expanded form: it will start with the first non-blank character and extend all the way to the end of the line (it may contain blanks, of course). Finally, after a single empty line, you will get a short text message, fitting on a single line, which you should then expand and wrap left-justified (i.e., every line of your output should be no longer than the text width, and every line should start with a non-blank character). So, you may have to drop a blank space, or even a few of them, at the front of a line to fit it in correctly.

Note that we will never put words in the message that are longer than the chat width—so no need to hyphenate!

Here's a short example, using a width of 20, and with a little "digit ruler" below to show that it works:

Input:

```
20
lol   laughing out loud
brb   be right back
:)    *smile*
<3    *heart*

Hey gramps! Great joke -- lol! Mom's calling, brb!
```

Output:

```
Hey gramps! Great
joke -- laughing out
loud! Mom's calling,
be right back!

12345678901234567890
```