Twenty-Fifth Annual
Willamette University—TechStart High School Programming Contest

Saturday 12 March 2011

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 to write a simple* **GUI** *interface, but you should accept the basic data as described.)*

1. Fraction subtraction

You're helping your little brother with his math homework by writing a program to generate problems for him and check his answers. This week he's learning fractions—more specifically, he has to do subtraction problems involving fractions with various numerators and denominators, but he also needs to express the answers in "lowest terms", i.e., so that the numerator is not evenly divisible by the denominator.

Your program should read in series of pairs of fractions, subtract the second fraction from the first, and return an answer for each pair in lowest terms. If needed, you may request a sentinel value of two 0/0 fractions to terminate the input (if you write a GUI we can just close the window). Note that the results of the subtraction may be negative: we'll stick to non-negative *inputs,* however, and we won't use zeros in the denominators, as the results would be undefined. Here are some sample inputs and outputs:

Input:	`2/3  3/5`	`Output:`	`1/15`
Input:	`7/8  12/5`	`Output:`	`-61/40`
Input:	`123/17  311/113`	`Output:`	`8612/1921`
Input:	`0/0  0/0`	*<program ends>*	

(Gosh, that second-to-last one might be tough for your little brother—but your program should handle it anyway!)

2. Bracket balancing

Programmers know how annoying it can be when parentheses, brackets and braces get out of balance in their code. You are working on a programmers' editor and need to write a utility which will highlight selected portions of the code in red if any of the bracketing characters within are out of balance. Your code will check the selection for "round" parentheses `()`, square brackets `[]`, curly braces `{}` and even angle brackets `<>`. Furthermore, you should check a "no crossings" rule, so that different shapes of brackets don't get mixed up. For example, a string like " `< () [ { ] } >` " is not allowed: even though the curly braces and the square brackets come in matched pairs, they cross each other in between. Finally, your program should ignore any *non*-bracket characters, so that other bits of code (perhaps variable names, numbers, and keywords) can be intermixed. As output, you should write out the word "`OK`" (if everything balances, with no crossings) or "`BAD`" if something goes wrong. (If you are using a GUI, you may even highlight the code in green or red to indicate `OK` and `BAD` strings … just make sure that it's clear when your check is finished!)

Some examples:

Input:	`foo(bar[i].{}<<>>[baz]123)`	`Output:`	`OK`
Input:	`oof{}[<<quux>> 17 ] ( {}`	`Output:`	`BAD`
Input:	`[17] () 23 .. < [ > foo ]`	`Output:`	`BAD`

3. Binary buddies

We can write out numbers in base two, or binary form, using just the digits `0` and `1`: for example, the number `37` is written out in binary as `100101`, whereas the number `41` is written in binary as `101001`.

You might have noticed that the binary form for `37` is the exact *reverse* of the binary form for `41`: when this happens, we call the two numbers "binary buddies". In some cases, we might have to add some leading zeros to one of the numbers to make the connection: for example, `11` and `52` are binary buddies, since `11` in binary form is `1011`, and `52` in binary form is `110100`.

Your program should read in a sequence of numbers (terminated by a zero, if you need it) and print out several lines in response: all the numbers in a given *output line* should be binary buddies of each other, and *every number from the input* should appear on some output line, even if it's only all by its lonesome … **sniff**).

For example, here is a sample input line followed by output lines that gather binary buddies together:

Input:
```
64 12 16 56 34 4 112 20 16 40 96
```

Output:
```
64 16 4 16
12 96
56 112
34
20 40
```

4. Code finder

Another little feature you want to implement for your programmer's text editor is a "coder's search": this should allow the user to find a specified string anywhere in their code, but *only* in the part of the text that really is *program code.* More specifically, while searching you should ignore any text that comes inside *strings or comments.* We'll have to chose a specific style of comments here, just to be definite, so we'll use the C language convention that comments are bounded by "`/*`" and "`*/`". Strings, on the other hand, will be set in the usual double quotation marks that are common to most languages.

Comment brackets and quotation marks will be balanced in the input texts we give you; comments may be nested, but there will be *no quotes inside quoted strings.* The code will always start out, at the beginning of the input, as normal code (not inside a comment or string), and will always come back to that same state at the end. Finally, remember that you also have to search for a target string along the way … just *not* inside any comments or strings!

As input, you will get (on two separate lines, say, or as two fields in a GUI) a target string to search *for* and a source string to search *in.* The target will never have comment brackets or quotation marks (whew!), but it may have other odd mixes of letters, digits, punctuation marks and spaces (but no tabs, returns or escape characters!). You should return the indices of *all occurrences* of the target string inside the source string, even overlapped ones, but not commented or quoted ones. Indices in the source string start at character 0 and continue up, one per character, including those in comments and quoted strings, until the end of the string. *(I have included some single-digit "index markers" in the example to help count—they won't be there in real inputs!)*

Input:
```
in
inner = "input interrogation" in /* check 5368 for error */ + inner ;
012345678901234567890123456789012345678901234567890123456789012345678
```

Output:
```
Target found at:  0  30  62
```

Input:
```
rrr
rr rrrrr "rrrrrr ** rrr "  rrr  /* arrrrgh! */ rrrr
012345678901234567890123456789012345678901234567890
```

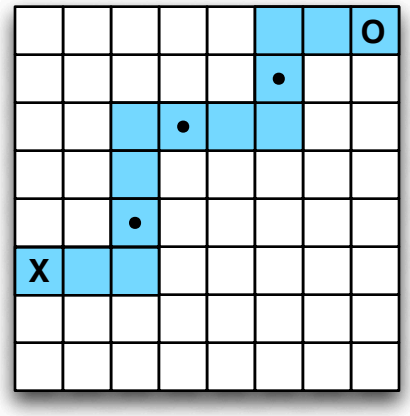
Output:
```
Target found at:  3  4  5  27  47  48
```

5. Knight moves

A knight on a chessboard can move either 2 squares up and one over, or one square up and two over, in any direction, in a single move. Let's number the squares of the chessboard so that position `(1,1)` is the lower-left corner and `(1,8)` the upper-right corner (for this problem, all our chess boards are 8 by 8).

Write a program which will read in the co-ordinates of a starting position for a knight on a chessboard, followed by the co-ordinates of a "goal" square on the same board. Your program should print out a sequence of squares representing a valid path, using only knight's moves, and always staying on the board, which will take the knight from the start to the goal (it doesn't need to be the shortest path, just any valid one will do). The "squares" in the inputs and the outputs will be pairs of numbers: you may request and print them with parentheses and commas, but we will assume no input punctuation unless you ask. *Note: you should include the start and the goal in your output!*

In this example diagram, **X** marks the start, **O** the goal and • the stopping points along the way:



Input: `1 3  8 8`

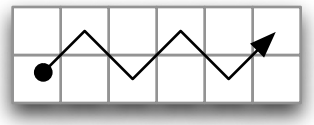
Output: `1 3  3 4  4 6  6 7  8 8`

6. Bouncing bishops

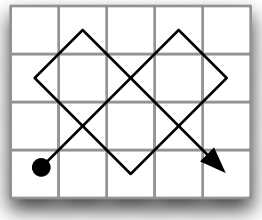
Consider a bishop on a chess board: he can move only along the diagonals, but in any of 4 directions. Now imagine that he starts on some square and heads off in a certain direction ... ooops! He hits the edge of the board and bounces off in a new diagonal direction! Ooops, he bounces again, off another edge, and so on, until one of two things happens: he either ends up stuck in a corner with no place to go, or he returns to his original position. Since he tends to get stuck a lot on a square chess board, we will make our chess boards rectangular in general, not just square.

Now we can ask, given a certain starting square and direction, **how many times will a bishop bounce off the walls before he either gets stuck in a corner or returns to his start?** Assume that the squares of a rectangular chess board are given integer co-ordinates, with the lower left being `(1,1)`. We will provide 3 pairs of integers on a line: the first pair describes the dimensions of the chess board, width and then height. The second pair describes the starting point of the bishop, measuring from the left edge and up from the bottom of the board. Finally, each of the last pair of numbers will be 1 or –1, indicating his initial direction, with 1 being up, –1 being down and 1 being right, –1 left.

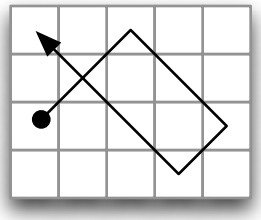
The bishop always bounces at 90° angles, and always occupies a whole square. He does not bounce back out of corners (i.e., he comes to a stop there), and we don't count hitting the corner as a bounce. Finally, we will never start a bishop right against a wall and traveling toward that same wall, as it is a little hard to see how (or even whether) he should bounce (starting against a wall and traveling *away* from that wall *is* allowed).



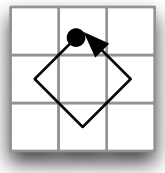
board: 6 x 2
start: (1,1)
direction: (1,1)
bounces = 4



board: 5 x 4
start: (1,1)
direction: (1,1)
bounces = 5



board: 5 x 4
start: (1,2)
direction: (1,1)
bounces = 3



board: 3 x 3
start: (2,3)
direction: (-1,-1)
bounces = 3

Given the input `3 3 2 3 -1 -1`, your program should return `3` bounces (this is the 4th example above).

7. Pizza splittings

One of the big pizza chains has hired you to help with the planning of their weekly specials: specifically, they want you to calculate the weight of a slice of pizza, given the size of the pizza, the number of slices and the weights of the toppings. Their pizzas come in four sizes, each with a fixed weight of dough in pounds: *large* (3 pounds), *gigantic* (5 pounds), *humungous* (7 pounds) and *prodigious* (13 pounds); we will use the initial letters to code the sizes, `L`, `G`, `H` or `P`. Each pizza may also have as many as 7 toppings, each with a weight that will be given in the input. Finally, each pizza is cut into a certain number of slices based on its size: `L`=6; `G`=8; `H`=10 and `P`=16. Your program should read in a line consisting of a single letter for the size of the pizza, and up to seven numbers (terminated by a `0.0` if needed) representing the weights of the toppings (the numbers will always have a decimal point and at least one leading digit, but no more than a couple of digits more). You may assume that the toppings are distributed evenly on the pizza, so that each slice gets an equal share of the weight of each item (not *my* experience, but oh well …). Finally, your program should verify the number of slices as part of the output. For example, a run of your program might look like this:

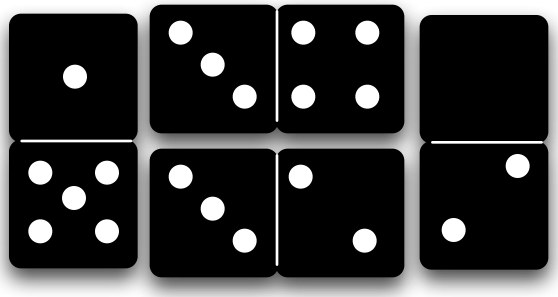
```
Input:  H  2.3  4.5  6.0  0.0
Output: 10 slices weighing 1.98 pounds each

Input:  P  1.7  8.3  5.25  3.72  2.0  4.8  0.0
Output: 16 slices weighing 2.42313 pounds each
```

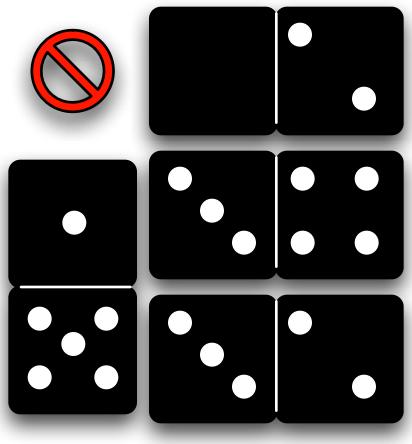
Your answers should be accurate to at least 2 decimal places (but you may print more places if you wish).

8. Domino fittings

Dominoes are little two-part rectangular tiles with a certain number of dots printed on each part. Say we have a set of dominoes lying on the table: we'd like to arrange them into a nice, "squared-off" rectangle, but according to the following rule: wherever two dominoes touch, in other words where they share an edge, the number of dots on the parts that share an edge should either *both be even* or *both be odd*. Of course, a single domino may have one even side and one odd side, so the rule applies only where *different* dominoes share an "external edge" between them. Just to make things easier, we'll allow dominoes with a blank side (zero dots) to be considered "wild cards" which can match either an odd or an even number, or both at the same time, i.e., on two or three different shared edges. Below is a picture of 4 dominoes arranged in the way we want, on the left, and a picture of the same dominos arranged in a bad way (because of the gap) on the right.



good solution



bad solution—gap in upper left

Your program should read in a set of dominoes as pairs of integer numbers, between 0 and 6 inclusive. If asked, we will place vertical bars between the joined sides of a single domino, and spaces between separate dominos, but you may also ask for just spaces, or for dominos on separate lines, or in a GUI form. Your output, however, should be on a console or in a text area using a monospaced font (such as Courier) so that everything fits neatly in a grid. For example, input and output for the situation on the left above might be given like this:

Input: `5|1  2|0  3|2  3|4`

```
Output:    1 3 4 0
           5 3 2 2
```

9. Sharing the spotlight

You are helping the stage crew at the Tony Awards to plan the choreography for the big, show-stopping song-and-dance numbers that should be the hit of the evening. These big ensemble productions will feature a number of celebrities working together, so of course the tensions are running high. The producer needs your help to keep some of those big egos in check: specifically, you need to make sure that none of the celebrities involved in the big dance numbers will have to share a spotlight with anyone else … *literally!*

Write a program which reads in a set of celebrity names and spot lights, and prints out pairs of celebrity names whose spotlights overlap. Each spotlight will be defined as a circle with a given center and radius: we will supply three floating-point numbers, one for the *x*-coordinate of the circle, one for the *y*-coordinate and one for the radius. The single celebrity name (Britney, Justin, Oprah, Charlie, etc.) given with each of the lights are just used to identify overlaps (names will be a single word consisting of upper or lowercase letters; they will come before the numeric data on the input lines, or you may use a GUI).

Note that the lights are to be considered overlapping even if they only just barely touch on the edge!

Here is an example of input and output:

Input:
```
Xavier  2.0 5.0 1.0    Warren 13.0 7.0 1.0  Oprah 7.0 4.0 3.0
Britney 4.0 7.0 2.0    Justin 10.0 7.0 2.0
```

Output: `Xavier overlaps Britney; Britney overlaps Oprah; Oprah overlaps Justin`

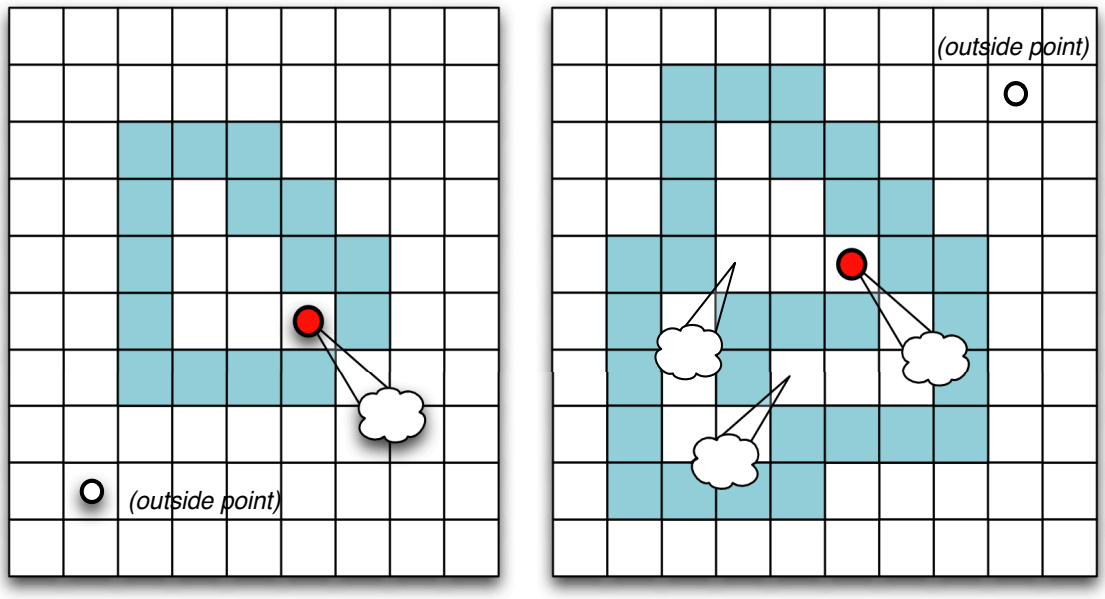
10. Containing the leaks

Bad news: you have been given the job of cleaning up after a nuclear reactor accident, over in Chernobyl's sister city, Djesknobyl. Fortunately, your team arrived after an initial containment sweep was made: robots were used to put some special lead-filled concrete blocks into place around the area according to a grid pattern. The blocks are supposed to shield you and your team from the deadly radioactive gasses that are leaking from the "hot spot" inside the reactor core.

Unfortunately, you have found a flaw in the plan the robots used in the initial containment sweep: if the concrete blocks are placed flush against each other, side-to-side, they form a tight seal. But when they are placed diagonally, so that they only meet at their corners, a leaky spot is created where the deadly gasses can escape (see the diagram below). And, given this "gap" in the robots' programming, you are concerned that they may have left even larger gaps in the containment wall … .

In the diagram on the left, below, you can see how radioactive gasses might leak from the circular "hot spot" into the area outside the containment wall. In the right-hand diagram, there are similar leaks, but notice how the solid, flush placement of the outer blocks means that no gas can ultimately escape. In order to protect yourself and your crew from danger, you need to write a program which will check whether gas can escape (as on the left) or not (as on the right). We will give you the coordinates of the radioactive "hot spot" and a second spot, guaranteed to be on the outside of the containment wall, as well as the coordinates of all the blocks. You should return a response of "`RUN AWAY!`" or "`SAFE`".

Your input will consist of *pairs* of positive integers, in the following order: hot spot, outside spot, and then some number of concrete blocks. You can request these be input one per line or in a GUI, but please make it easy for us to paste in larger inputs, as we may need 30 or more blocks for some complicated situations!



The example on the left above would be noted like this, although we have only shown the first few blocks:

Input: `6 5  2 2  3 4  4 4  5 4  6 4  3 8  4 8  5 8  7 5 ...`

Output: `RUN AWAY!`

Note that the blocks may be provided in an arbitrary *order!*

11. Inversion conversion I

Some mathematical functions have *inverses,* functions that "undo" what the original function did. For example, a function that adds 3 to its input has, as its inverse, a function that *subtracts* 3 from its input: we'll write these two functions as `(+3)` and `(–3)`, including the parentheses to emphasize that the function input is the "missing part". A function that multiplies its input by 4 has, as its inverse, a function which divides its input by 4: we'll write these as `(*4)` and `(/4)`. Of course, if we have a subtraction or a division, rather than addition or multiplication, *which side* the number is on matters: the `(7–)` function and the `(–7)` function are different, since one subtracts its input *from* 7 and the other subtracts 7 *from its input.*

Of course, more interesting functions have more interesting inverses. Let's agree to focus on certain "compound functions" built from these simple ones. We can combine a function like `(+3)` and a function like `(*4)` to get a function which first adds 3 to its input, and then multiplies the result by 4: we'll write the combination as `(+3)(*4)`, i.e., first add 3, then multiply by 4. More complex examples might include three or more simple functions combined, one after the other, working from left to right.

We will give you, as input, a line describing a compound function written in this way, followed by a list of floating-point numbers (a couple of digits after the decimal point at most): you should print out the results of applying the *compound function's* **inverse function** *when applied to those arguments.* Note: that means you have to figure out the original compound function, then figure out its inverse, then apply the inverse!

Input: `(7-) (*4) (+3)` *this means:* `f(x) = ((7-x)*4)+3`

`1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0`

Output: `7.5 7.25 7.0 6.75 6.5 6.25 6.0 5.75 5.5 5.25`

Input: `(/6) (5–) (*2)` *this means:* `f(x) = (5–(x/6))*2`

`2.0 4.0 6.0 8.0 10.0`

Output: `24.0 18.0 12.0 6.0 0.0`

(We won't really input the "this means" *part. What are the relevant inverse functions? Try it by hand and see!)*

12. Inversion conversion II

OK, it's time for the big leagues in function inversion! Say that we have an arithmetic expression written out in terms of constants (i.e., numbers) and binary operators, including addition (`+`), subtraction (`–`), multiplication (`*`), division (`/`) and powers (exponentials), for which we'll use the "hat" symbol, (`^`). *(Be careful: in some languages that symbol means something completely different!)* Assume furthermore that there is *exactly one occurrence* of a variable "`x`" somewhere in the expression. (We will use fully-parenthesized expressions, so that you don't have to worry about the order of operators.) This expression will determine a function of *x*, namely the one you would get if you put the input value in place of *x* and evaluated the expression.

Just as for the simpler compound functions from the last problem, each function we can write this way will have a unique inverse function, as long as we assume that nothing bad happens (like division by zero) and we agree to use positive numbers when there is an ambiguity (for example, we will use 2 for the square root of 4, never –2). What are the relevant inverses for the power function? It depends on which side the "number part" is on: you may need square roots, cube roots, etc., but also *logarithms in various bases.*

As for the last problem, we will give you a "function expression" on one line and then a series of values on the next: you should compute the inverse function and apply it to all of the values in turn.

Input:
```
(3 * (x + 2))
10 20 30 40 50
```

Output:
```
1.333 4.667 8.000 11.333 14.667
```
— *rounded to three places*

Input:
```
(((2 * 4) + (x ^ 3)) - (7 * 3))
100  200  300  400  500
```

Output:
```
4.8346  5.9721  6.7897  7.4470  8.0052
```
— *rounded to four places*

13. Retro game decoder

Your friend Tim loves to play retro computer text games, the kind where you walk around in a "dungeon", finding treasure and defeating monsters. He's been frustrated lately, however, because he just can't beat this particular game: it's a "community map", i.e., a dungeon level created by a game fan. He wants you to help him "peek" at the internals of the map to see what keeps tripping him up. Unfortunately, the fan who designed this 3rd-party add-on used a map-editing system that compiles the data down to a form which makes it (purposely) hard for people to ~~cheat~~ I mean, *read*. So, he wants you to write a decoder that will display the hidden information that defines the map for him.

What you've discovered is that the map is encoded using a combination of punctuation marks to indicate rooms, monsters, treasures and weapons, along with English text (lowercase letters and spaces only) used to write out descriptions to the player. The encoding seems to show that the map is a series of rooms, each of which contains some number of loose items, and monsters, who may themselves be holding items. The items in turn may be treasures, potions or weapons.

Each room code starts with a "`#`" sign, which is followed by codes for monsters or items, intermixed, then some description text (lowercase letters and spaces), then possibly by more room codes. Each monster code starts with a "`*`": it may have some (or none) items following it, and then a description text. The items are just single-character codes, but may be repeated any number of times to indicate several items of the same kind: they include "`$`" for treasure, "`!`" for weapons (looks like a dagger to me) and "`?`" for potions (always a mystery!). Your program should read in a series of codes for rooms, monsters, and items and print out a report for your friend Tim as follows: for each room, print a number (start at 1 and count up) followed by its text description; then indent and report how many "loose" treasures, potions and weapons are in the room. Then list each monster in the room, indented underneath the room. Number the monsters starting from 1 within each room, and print their description and the number of each kind of item they hold.

For example, you might write out the report below for this coded input; note that the first room contains some loose items listed *before* the gargoyle monster, as well as two loose items *after* the monster:

Input:
```
#$!*$$$gargoyle$?dark and damp#$$?*!big orc*!little orc$$?dank and foul
```

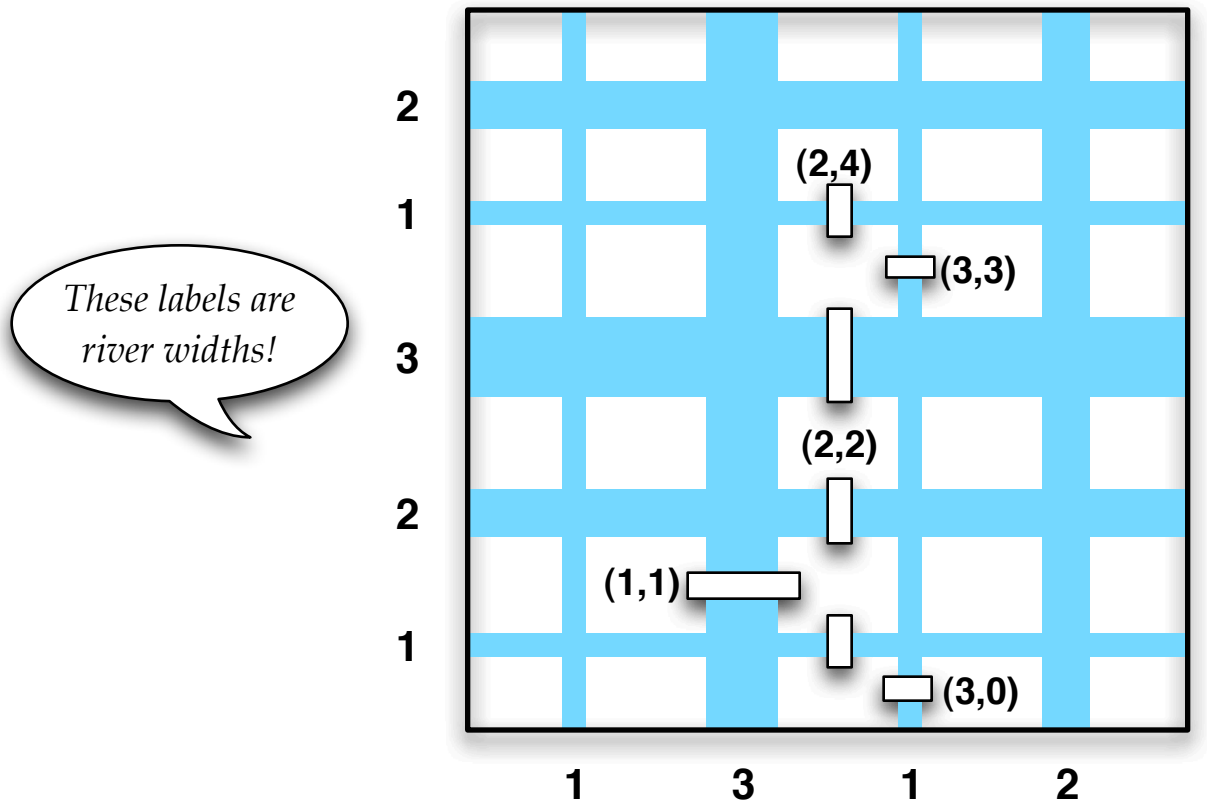
Output:
```
Room 1: dark and damp
     2 treasure(s), 1 weapon(s), 1 potion(s)
     Monster 1: gargoyle; 3 treasure(s), 0 weapon(s), 0 potion(s)
Room 2: dank and foul
     4 treasure(s), 0 weapon(s), 2 potion(s)
     Monster 1: big orc; 0 treasure(s), 1 weapon(s), 0 potion(s)
     Monster 2: little orc; 0 treasure(s), 1 weapon(s), 0 potion(s)
```

14. Building bridges

After finishing school you decide to join the Peace Corps and help the world become a better place. Your particular assignment puts you in a small island country in the south Pacific with an interesting geography: the island is criss-crossed by rivers of various widths, all of which run almost exactly east-west or north-south. This effectively divides the island into a grid of open areas bounded by rivers—some of these open areas are occupied by small villages. You and your trusty crew of civil engineers need to build enough bridges so that the people from *any* village will be able to get to any *other* village on the island via bridges.

On the sample map below, we have shown where the villages are by placing their grid coordinates in the corresponding open space. Grid spaces start at `(0,0)` in the lower left; we use the left and right coordinates for horizontal (x) and vertical (y) in the usual fashion. *We've also marked the widths of the rivers along the lower and left axes.* Finally, we've put in some sample bridges that solve the village connection problem.

The trick is, you only have a certain amount of bridge material to work with—the cost in materials of building a bridge is directly proportional to the width of the river, so that a 3-wide river costs 3 units of material, etc. In the sample map, the seven bridges built will have used up a total of 12 units of material. Other bridge layouts are clearly possible to make the needed connections, but some would use more materials, and others perhaps even less. In any case, the problem you must solve is to find *some* working set of bridges using an available supply of materials, or report that there are not enough to do the job.



Your input will consist of a material supply amount, two lines of river widths (x-axis first, then y-axis) and then a series of pairs of numbers representing village coordinates (if you need it, you can request a sentinel pair with -1 values to terminate the list). All numeric values will be non-negative integers. You don't need to read in parentheses and commas for the coordinate pairs, but be careful about where the pairs "break". Your output should report, for each bridge you build, two co-ordinate pairs (for the two grid areas between which the bridge should be built) and a cost for that bridge (i.e., the width of that river), just as a check. The total of your costs should be less than the amount of material you have, or you should write out "`No solution possible`" if there is no way to build the required bridges.

For example, input and output for the island shown above would be as follows:

Input:
```
15
1  3  1  2
1  2  3  1  2
1 1   2 4   3 3   3 0   2 2
```

Output: `bridge from 1 1 to 2 1 (3 units); bridge from 2 1 to 2 0 (1 unit); ...` *(etc.)*