UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

Programming Contest **P. N. Hilfinger**
Fall 1993

Programming Problems

You have 5 hours in which to solve as many of the following eight problems as possible. Put each complete solution into a single N`.c` file, where N is the number of the problem. Each program must reside entirely in a single file; it must not `#include` any non-standard files.

Aside from files in the standard system libraries, you may not use any pre-existing computer-readable files to supply source code; you must type in everything yourself. You may not use utilities such as `yacc`, `bison`, `lex`, or `flex` to produce programs. Your programs may not create other processes (as with the `system`, `popen`, `fork`, or `exec` series of calls). You may use any inanimate reference materials you desire, but no people. You can be disqualified for breaking these rules.

When you have a solution to problem number N that you wish to submit, use the command

`submit` N

from the directory containing N`.c`. Before actually submitting your program, `submit` will first compile it and run it on one sample input file. No submission that is sent after the end of the contest will count. You should be aware that `submit` takes some time before it actually sends a program. In an emergency, you can use

`submit -f` N

which submits problem N without any checks.

You will be penalized for incorrect submissions that get past the simple test administered by `submit`, so be sure to test your programs. All tests will use the compilation command

`contest-gcc` N

followed by one or more execution tests of the form (Bourne shell):

N `<` *test-input-file* `2>` `/dev/null | diff -b -` *test-output-file*

That is, output to `stderr` is ignored, and in comparing output, leading and trailing blanks are ignored and sequences of blanks are compressed to single blanks. Otherwise, the comparison is literal; be sure to follow the output formats *exactly.* Each test is subject to a time limit of about 15 seconds. You will be advised by mail whether your submissions pass.

The command `contest-gcc` *optional-switches* N, where N is the number of a problem is available to you for developing and testing your solutions. It is equivalent to

`gcc-2.4.5 -Wall -o` N *optional-switches* `-I`*our-includes* N`.c -lm`

The *optional-switches* are `-O` (optimize) and `-g` (include debugging information). The *our-includes* directory contains some of the standard header files modified to have prototypes, so that `gcc` will catch more errors. We suggest that you use this command and start each of your source files with

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>
```

All input will be placed in `stdin`, and all output must be produced on `stdout`. Anything written to `stderr` will be ignored. Your program *must* exit with a status code of 0 (use `exit(0)`). Because the testing of programs is automatic, you should be careful to adhere closely to the output format described. Make sure that the last line of output ends with a newline. You may assume that the input conforms to any restrictions in the problem statement; you need not check the input for correctness. Consequently, you are free to use `scanf` to read in numbers and strings.

Scoring. Scoring will be according to the ACM Contest Rules. You will be ranked by the number of problems solved. Where two or more contestants complete the same number of problems, they will be ranked by the *total time* required for the problems solved. The total time is defined as the sum of the *time consumed* for each of the problems solved. The time consumed on a problem is the time elapsed between the start of the contest and successful submission, plus 20 minutes for each unsuccessful submission, and minus the time spent judging your entries. Unsuccessful submissions of problems that are not solved do not count. As a matter of strategy, you can derive from these rules that it is best to work on the problems in order of increasing expected completion time.

Protests. Should you disagree with the rejection of one of your problems, first prepare a file containing the explanation for your protest, and then use the `protest` command (without arguments). It will ask you for the problem number, the submission number (submission 0 is your first submission of a problem, 1 the second, etc.), and the name of the file containing your explanation. Do not protest without first checking carefully; groundless protests will be result in a 5-minute penalty (see Scoring above).

Terminology. The term *free-form input* indicates that input numbers, words, or tokens are separated from each other by arbitrary whitespace characters. By standard C/UNIX convention, a whitespace character is a space, tab, return, newline, formfeed, or vertical tab character.

1. You are given a set of opaque rectangles in three-dimensional space and asked to find whether a point-sized observer located at one point in that space can see another observer located at a second point. Each rectangle is either vertical (upright) and runs north-south or east-west, or is horizontal (parallel to the ground). The data comprise

- A positive integer N giving the number of rectangles.
- A sequence of $6N$ floating-point numbers giving the x, y, and z coordinates of two opposite vertices of each of N rectangles. Each rectangle edge will be parallel to an axis, so such a vertex pair uniquely determines a rectangle. As further consequences, each vertex pair will have the same coordinate along either the x, y, or z axis (the axis may differ from one rectangle to another) and each rectangle will lie in a plane parallel to the xy, xz, or yz planes.
- A sequence of $6K$ floating-point numbers, for some $K \geq 0$ (which is not explicitly given). Each group of 6—$x_0, y_0, z_0, x_1, y_1, z_1$—gives the coordinates of two observation points, (x_0, y_0, z_0) and (x_1, y_1, z_1).

The data are free-form. The rectangles may intersect each other. Two observer points are visible to each other if and only if their line of sight is not blocked by any rectangle. A line of sight is blocked by a rectangle if and only if neither observer lies in the same plane as that rectangle and the line segment joining them intersects the interior of the rectangle. You may assume no lines of sight will intersect edges unless one of the observers is actually on that edge.

The output of the program will consist of a sequence of K lines, each having one of the two forms

`(`x_0`,  `y_0`,  `z_0`) is visible from (`x_1`,  `y_1`,  `z_1`).`

or

`(`x_0`,  `y_0`,  `z_0`) is invisible from (`x_1`,  `y_1`,  `z_1`).`

where all the coordinates are printed using the C `printf` format `"%.2f"`, leaving at least one space after each comma.

For example, suppose the input is as follows.

```
5
0 0 0         1.5 2.25 0
3.5 2.25 0    2.5 0 0
0 2.5 0       3.5 2.25 0
2.5 0.75 0    3.5 0.75 2
0 0 -4        0   2.5  4

1.5 0 2   -3 1.25 1.5
3 2 0.5   1.5 0 2
3 0 2     1.5 2.5 -2
2.5 0 2   1.5 2.5 -2
```

This represents the situation shown in Figure 1. The output will be as follows.

```
(1.50, 0.00, 2.00) is invisible from (-3.00, 1.25, 1.50).
(3.00, 2.00, 0.50) is visible from (1.50, 0.00, 2.00).
(3.00, 0.00, 2.00) is invisible from (1.50, 2.50, -2.00).
(2.50, 0.00, 2.00) is visible from (1.50, 2.50, -2.00).
```

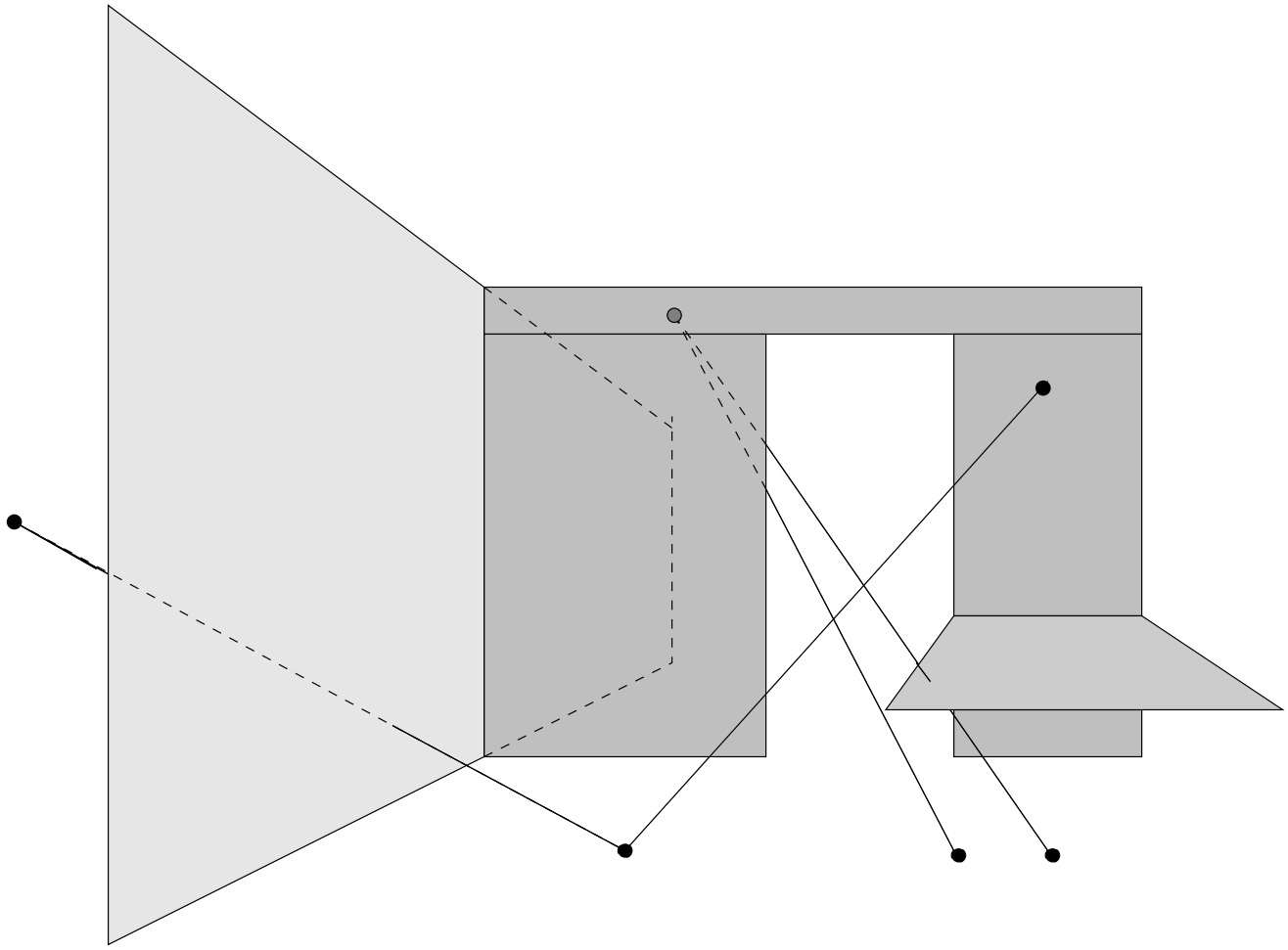

Figure 1: Rectangles and observation points described by the sample input. The positive z axis points up out of the page. The diagram also shows the lines of sight between the observation points. Hidden lines are dashed.

2. Given a set of pieces allegedly from a jigsaw puzzle, solve the puzzle. The pieces consist of small squares (with sides one unit long) glued together along their edges. The input consists of zero or more sets of data. Each set of data consists of

- Positive free-form integers W, H, and P. These stand for the width and height of the puzzle (in units), and the number of pieces ($P \leq 26$), respectively.
- A sequence of P *piece descriptions*, each consisting of
  - Two positive free-form integers W_i and H_i, the width and height of the piece in units.
  - H_i lines, each containing exactly W_i periods and asterisks. The first and last lines each contain at least one asterisk, as do the first and last columns. These lines represent the piece; the asterisks stand for solid unit rectangles, and the periods for spaces.

For the purpose of output, the pieces of the puzzle are labeled 'a' through 'z' in the order they are specified.

For each set of data in the input, you are to output a line identifying the data set, a copy of the input, and a solution of the puzzle, if one exists, or a statement that no solution exists. Place a blank line after each set of output. The format is as shown in the example on the next page.

To be a solution, the pieces must fit within the specified W by H space without overlapping, rotating, or flipping, and must fill the space entirely. When there is more than one solution, the solution you should produce is the one that places each piece in a "better" location than all pieces labeled with letters that occur later in the alphabet. One location for a piece is "better" than another if it is higher, or if it is at the same level and more to the left. For example, according to this rule, the solutions

```
aabb     bbbcc    aaa    ab
bbcc     baacc    bbb    ab
```

are respectively preferred to

```
ccbb     ccbbb    bbb    ba
bbaa     ccbaa    aaa    ba
```

The next page contains a sample set of input and the desired output.

The input consists of three sets of data.

```
4 4 4
2 2
*.
**
3 3
***
.**
..*
1 2
*
*
4 2
****
...*

4 3 2
2 4
*.
**
**
**
4 2
...*
****
4 3 2
4 2
**.*
.***
1 1
*
```

The output for this input file looks like this.

```
Puzzle #1 measures 4 by 4, with 4 pieces:

Piece a:
a.
aa

Piece b:
bbb
.bb
..b

Piece c:
c
c

Piece d:
dddd
...d

Solution:
dddd
bbbd
abbc
aabc

Puzzle #2 measures 4 by 3, with 2 pieces:

Piece a:
a.
aa
aa
aa

Piece b:
...b
bbbb

There is no solution.

Puzzle #3 measures 4 by 3, with 2 pieces:

Piece a:
aa.a
.aaa

Piece b:
b

There is no solution.
```

3. It is often useful to know when a given subprogram calls another subprogram. Write a program that takes as input a pre-processed C program and produces an approximate list of the functions within it and the functions they call.

The "approximation" alluded to refers to the following substantial simplifications of C syntax for the purposes of this problem:

A. Any sequence of the form

function-name <optional whitespace> (

occurring outside any pair of curly braces ('{}') or parentheses will be taken as the definition of a function named *function-name* (any valid C identifier, beginning with a letter or underscore and consisting of letters, digits, and underscores). Here, <optional whitespace> refers to any sequence (possibly empty) of blanks, tabs, and newlines.

B. Any identifier occurring somewhere inside a pair of curly braces will be taken as a call on a function, if there is a definition of that function (as defined in A) elsewhere in the input (possibly later).

C. Any call (as defined in B) is assumed to occur inside the definition of the most recent preceding function definition (as defined in A).

D. Anything inside a string constant (which runs from a quotation mark to the next quotation mark that is not preceded by a backslash) or a character constant (which runs from an apostrophe to the next apostrophe that is not preceded by a backslash) does not count as an identifier.

E. The character (if any) immediately before an identifier is always whitespace or a punctuation mark. Identifiers are never more than 1023 characters long.

F. Because the input is pre-processed, there are no comments, and anything starting with '#' up to the end of a line should be ignored (treated as if it were entirely whitespace).

G. Assume the input is from a valid C program. In particular, curly braces and parentheses are balanced.

H. A function may be defined twice (according to item A), in which case you should treat the second and subsequent definitions as continuations of the first (that is, add the calls following these later definitions to those already present).

Your output will consist of a sequence of lines of the form

$$G \text{ -> } F_1,\ F_2,\ \cdots,\ F_n$$

where G and the F_i are identifiers naming functions defined somewhere in the input (according to rule A above) and the function G contains calls of each of the F_i according to rule B above. Include only lines for which $n > 0$. There should be no duplicates among the F_i for each list. The output lines should be printed in the order in which the first definitions of the functions G appear in the input.

Likewise, the F_i in each list of callees should be ordered by the same rule. (Thus, no F_i is listed unless there is a definition for it in the input.) Place a blank line after each line of output.

For example, for the following input file,

```
# 1 "set3-0-.c"
extern int printf(char*, ...);

void h(void) {
    puts("Hello, printf\n");
}

int f
(int x) {
    x = x+1;
    if (x > 0) {
        g(x, printf);
    }
}

static void g (int h, int (*q)()) {
    q(f(h-3));
}

int main ()
{
    int z;
    f(42);
    h(13);
}
```

The desired output is as follows.

```
f => printf, g

g => h, f

main => h, f
```

As you can see, the output mistakenly claims that g calls h, even though it doesn't. Nevertheless, this *is* correct output, since the problem only calls for an approximation.

4. [With thanks to Paul Black.] Given a positive number base, B, and a set of base-B digits, S, we'll say that a positive integer is *regular* with respect to B and S if its base-B representation (ignoring leading 0's) contains only digits in S, and its square root is an integer with the same property. Write a program that inputs B and S and reports all regular integers with respect to B and S in the range 1 to $2^{32} - 1$ (yes, 2^{32}, not 2^{31}).

The input will be free-form and will consist of

- A decimal integer numeral B, with $1 < B \leq 36$.
- A sequence of base-B digits separated by whitespace. The possible digits are 0–9, followed by a–z ('a' stands for 10, 'b' for 11, etc.). Base-B digits are all less than B.

The output will consist of an echo of the input, plus a list of positive numbers (written in base B) and their square roots in the format illustrated by the following example. For the input

```
10
2 4 6 8
```

the output should be

```
For base 10 and digits 2, 4, 6, 8, the regular integers are
4 (2)
64 (8)
484 (22)
4624 (68)
68644 (262)
446224 (668)
44462224 (6668)
```

5. Consider a set of points in two dimensions. Assuming that there are at least two points in the set, each point has at least one nearest neighbor. The problem is to find a nearest neighbor for each point in the set.

The input will consist of a free-form sequence of pairs of coordinates, with each coordinate being an integer in the range -2^{14} to $2^{14} - 1$. You are to produce a sequence of lines of the form

$$N \texttt{ -> } M_1 \;\; M_2 \;\; \cdots$$

where $N, M_i > 0$ are the numbers of points; the first pair of coordinates in the input being those of point #1, the second pair being those of point #2, and so forth. This line will mean that the nearest neighbors to point N are the points M_i (that is, all the M_i are at the same distance from N and this is as close as any point comes to N). The lines are listed in order of strictly increasing N and on each line, the M_i are listed in strictly increasing order.

The input data will contain no duplicates. You may assume that there are no more than $11,000$ points in the data. At least one of the test data sets will have nearly this many points.

For example, given the following input (illustrated in Figure 2),

```
0 0
5 7
9 1 6 6 10 -1 12 0
```

The output would be

```
1 -> 4
2 -> 4
3 -> 5
4 -> 2
5 -> 3 6
6 -> 5
```

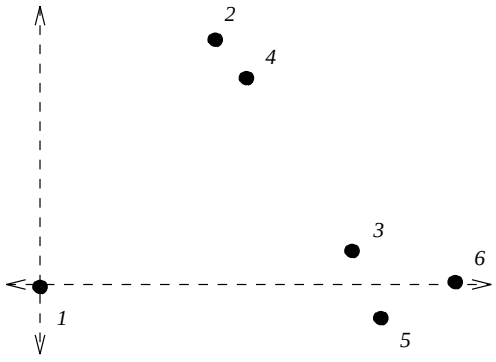

Figure 2: A sample input consisting of six points.

6. The huge and popular Twisty Maze Hotel has the unusual property that a clever arrangement of revolving doors makes all its corridors one-way; one has to use another corridor(s) to return to a room one has just left. Tonight, a private party is taking place in one subset of its rooms. Nobody will be allowed into these rooms without a reservation. The host feels that courtesy demands that he place greeters at the heads of strategic corridors to warn anyone who wants to enter those corridors that they will need to show their reservations when they get to the party, so that they don't get trapped with the party impassibly between them and the exit. He decides that he wants the following constraints to be satisfied.

- Greeters should be placed only at the heads of corridors through which it is inevitable that one will reach the party (that is, from which there is no way get out of the hotel except through one of the rooms occupied by the party).
- To give hotel guests who are not going to the party a chance to find an alternate route, greeters should be placed so that guests are warned when they *first* try to enter a corridor that leads inevitably to the party.
- Consistent with the above constraints, there should be as few greeters as possible.

You are to write a program to help in the greeter-placement problem. The input will consist of the following free-form items.

- A positive integer, N, indicating the total number of rooms in the hotel.
- A non-negative integer, P, indicating the number of rooms reserved for the party.
- A sequence of integers, p_i, in the range $1 \leq p_i < N$ indicating the numbers of the rooms in which the party is being held.
- A sequence of pairs of distinct integers—R_0 and R_1—both in the range 0 to $N-1$, inclusive, indicating that there is a corridor running from room number R_0 to room number R_1. You may assume that no pair is repeated.

Room 0 is the hotel lobby. All hotel guests enter and exit the hotel from the lobby. The number of rooms, N, may be arbitrarily large. You may expect, however, that the number of corridors is no more than $4N$. You may assume that all rooms can be reached from the lobby, and that the lobby can be reached from all rooms (although, of course, one might have to crash the party to do it).

For example, the following describes the situation shown in Figure 3.

```
13
2 10 11
0 1     1 2 1 4 1 10
4 0     5 0
12 6    6 4 6 4
10 11   11 10 11 12
8 9     9 10
7 8     8 2
2 3     3 2 3 7
```

The output desired for this input is the following.

```
Place 2 greeters at the corridors leading from room(s)
    1 to 2
    1 to 10
```

Corridors are to be listed in increasing order of the room at the start of the corridor, and then (for the same starting room) in the order of the destination room number.

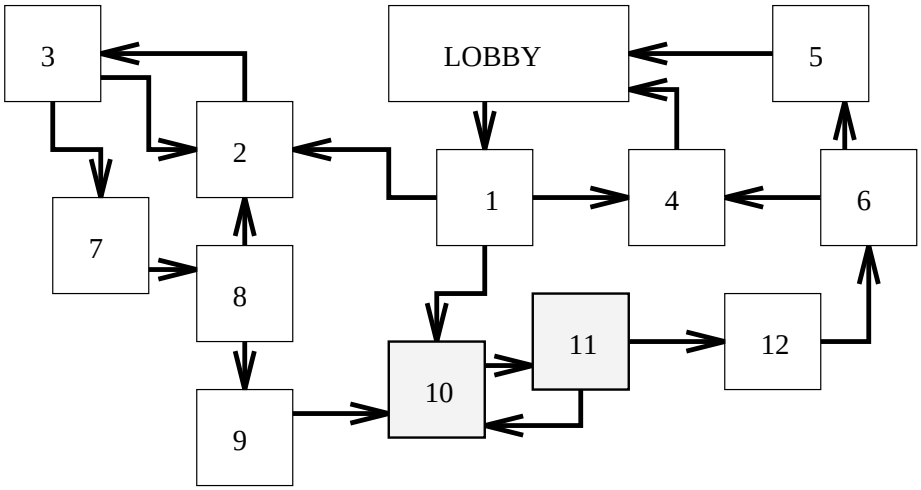

Figure 3: Diagram of hotel described by the input. Arrows show the directions of corridors. The party is in the shaded rooms.

7. You've probably seen those "cryptarithmetic" puzzles, in which one gets problems like these

```
  hello         too          i
+ there      + much     +  am
-------      ------     -----
  world        beer       not
```

and is asked to assign digits to each letter so that the resulting addition is correct. Each digit from 0 to 9 must be used at most once, and the leading digits may not be 0. In the above cases, for example, we can get the solutions

```
  56442         611          8
+ 15606      + 7942     +  97
-------      ------     -----
  72048        8553       105
```

and other solutions are possible, as well.

Write a program to find a solution to cryptarithmetic problems for which the input consists of triples of strings each containing up to 128 lower-case letters and the output is in the form given in the sample below.

For the input	Produce the output

`hello there world`
`too`
`much beer i am nuts`
	`  hello      56442`
	`+ there    + 15606`
	`-------    -------`
	`  world      72048`
	
	`   too       611`
	`+ much    + 7942`
	`------    ------`
	`  beer      8553`
	
	`     i`
	`+   am`
	`-------`
	`  nuts    No solution`

Precise horizontal spacing is not important; however, there must be a blank line after each problem, and the line of dashes must be two longer than the sum. You may assume the sum is at least as long as each addend. When multiple solutions are possible, prefer the one with the smallest units digit in the first addend. If there is more than one such solution, prefer the one with the smallest units digit in the second addend. If there are still multiple solutions, prefer the one with the smallest digit in the 10's place of the first addend, and so forth.

8. Expressions that use the operations of addition, negation, subtraction, exponentiation by a positive integer, and multiplication, together with non-negative integer literals and a set of variables, can denote any polynomial with integer coefficients in that set of variables. Write a program that, given two such expressions, reports whether they are identical, in the sense of having the same values for all possible integer values of the set of variables.

For simplicity, the input expressions will be presented free-form in prefix, with `+`, `-`, `*`, `^`, and `_` respectively representing addition, subtraction, multiplication, and unary negation. The set of variables will be the lower-case letters `a`–`z`. Integers (all non-negative) will be in the usual C format. Integer literals will be separated from variables and from each other either by operators or whitespace. For example, to represent the expressions

$$-3x(y-17)+2xy(18x^2+12), \quad -3xy+36x^3y+24xy+51x, \quad 2x-3y, \quad \text{and } 2x+z,$$

we can write

```
+ _** 3 x -y    17
  ***2 xy+*18*x x 12
+++ _ ** 3 x y **36 ** x x x y **24 x y * 51 x
- * 2 x * 3 y      + * 2 x z
```

Your program should accept pairs of expressions, echo them with parentheses inserted before each operator and after its operands, and print whether they are equal, using the format illustrated below. For the sample input given above, the output should be

```
(+ (_ (* (* 3 x) (- y 17))) (* (* (* 2 x) y) (+ (* 18 (* x x)) 12)))
  and
(+ (+ (+ (_ (* (* 3 x) y)) (* (* 36 (* (* x x) x) y))) (* (* 24 x) y)) (* 51 x)
  are equivalent.

(- (* 2 x) (* 3 y))
  and
(+ (* 2 x) z)
  are not equivalent.
```

Be sure to place blanks after each operator and after each operand that is not immediately followed by a right parenthesis. Your program should be prepared to accept any number of pairs of expressions. Place a blank line after every corresponding set of output, as shown.

Unless you choose an odd algorithm, you may generally assume that integer coefficients are all relatively small.