# 1 Computer Networks

You are designing a computer network to control trading on the international currency market. Because hundreds of millions of dollars, euros, and yen are being traded each day, you must ensure that data is transmitted correctly between computers.

One technique for ensuring data is being transmitted correctly is to compute and send a *checksum*, a small integer value representing the data transmitted. The receiving computer can then also compute its own checksum. If the checksums match, the data is probably correct. The IP (internet protocol) algorithm for computing a checksum is treat the data as a series of 16-bit numbers, add the numbers using one's complement arithmetic, then take the one's complement of the result.

In your network, data is being sent in *packets* of four 8-bit numbers (i.e., numbers between 0 and 255). The fourth number in each packet is a checksum for the packet. Your job is to compute a 8-bit checksum for each packet and verify the packet is correct. You can compute a checksum as follows:

1. Calculate $X$, the sum of the first three 8-bit numbers in the packet.
2. While $X \geq 256$, subtract 256 from $X$.
3. Subtract $X$ from 255.

The resulting value for $X$ should be a 8-bit number. Compare the checksum calculated with the checksum transmitted in each packet to verify transmission worked properly. Print out a warning for invalid data packets so the currency traders can resend data. The fate of the world economy is in your hands, so be careful!

## Input Format

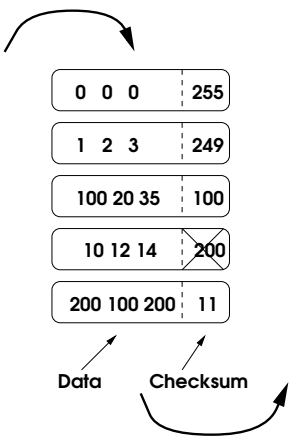
The input will consist of a number of packets, each on a separate line. Each packet will consist of four 8-bit numbers, where the fourth number is the checksum transmitted. The list of packets will be terminated by a line containing only $-1$.

## Output Format

For each packet, output "valid" or "invalid".

## Example

| **Input:** | **Output:** |
|---|---|
| 0 0 0 255 | valid |
| 1 2 3 249 | valid |
| 100 20 35 100 | valid |
| 10 12 14 200 | invalid |
| 200 100 200 11 | valid |
| -1 | |



## 2 Sequence of Differences

Consider a circular list with four integers, e.g., (0,1,4,11). For small non-negative integer values, repeatedly taking the absolute values of the differences of adjacent numbers eventually results in a list with identical values in each position.

```
(0,1,4,11) -> (1,3,7,11) -> (2,4,4,10) -> (2,0,6,8) -> (2,6,2,6) -> (4,4,4,4)
```

In this case, we needed five steps to reach a list with identical values. We could have started with smaller integer values and still needed five steps to reach a list with identical values.

```
(0,0,1,3) -> (0,1,2,3) -> (1,1,1,3) -> (0,0,2,2) -> (0,2,0,2) -> (2,2,2,2)
```

This sequence of five steps started with a list of non-negative integer values, each of which was less than or equal to 3. However, if all the list values were less than 3, no sequences of five steps would occur.

You are to write a program that computes the smallest maximum integer value needed in order to ensure that a sequence of N steps is used to reach a list all of whose elements have identical values. To make the problem simpler, you may assume N is 10 or less.

For example, to ensure a sequence of 3 steps, the value 1 is needed. The value 0 is not sufficient, since the sequence (0,0,0,0) already has identical values. The value 1 is sufficient, since we can find a four-member list consisting of 0's and 1's which will require 3 steps to reach identical values.

```
(1,0,0,0) -> (1,0,0,1) -> (1,0,1,0) -> (1,1,1,1)
```

In comparison, to ensure a sequence of 4 steps, the value of 1 is not sufficient since all four-member lists consisting of 0's and 1's will reach identical values in 3 or fewer steps.

### Input Format

The input will be the value N, the number of steps needed to reach identical values.

### Output Format

The output should produce one line of output containing N, the number of steps, and the smallest maximum integer value needed to create a four-member list which requires N steps to reach identical values.

### Examples

| **Input 1:** | **Output 1:** | **Input 2:** | **Output 2:** |
|---|---|---|---|
| 3 | 3 1 | 5 | 5 3 |

# 3 Sparse Vectors

A vector is a one-dimensional array of elements. The natural C++ implementation of a vector is as a one-dimensional array. However, in many applications, the elements of a vector have mostly zero values. Such a vector is said to be sparse. It is inefficient to use a one-dimensional array to store a sparse vector. It is also inefficient to add elements whose values are zero in forming sums of sparse vectors. Consequently, we should choose a different representation.

One possibility is to represent the elements of a sparse vector as a linked list of nodes, each of which contains an integer index, a numerical value, and a pointer to the next node. Generally, the entries of the list will correspond to the non-zero elements of the vector in order, with each entry containing the index and value for that entry. (This restriction may be violated if a zero value is explicitly assigned to an element).

Your goal is to write a program to add pairs of sparse vectors, creating new sparse vectors. The results of addition should not include any elements whose values are zero. You should then print the resulting vectors with elements in ascending order of index (from smallest index to largest).

## Input Format

The input will be several pairs of sparse vectors, with each vector on a separate line. Each sparse vector will consist of a number of index-value pairs, where the first number in each pair is an integer representing the index (location), and the second number is a floating-point number representing the actual value. You may assume all index locations are non-negative. Elements will be entered in ascending order of index. The list of vectors is terminated by a line containing only $-1$.

## Output Format

The output will be sparse vectors representing the sum of each pair of input vectors, each on a separate line. Vector elements should appear as pairs of indices and values, separated by a comma and a blank and enclosed in square braces. Vectors should appear as lists of elements separated by commas. The vector elements must be printed in ascending order of index. Vectors with no elements should appear as the string "empty vector".

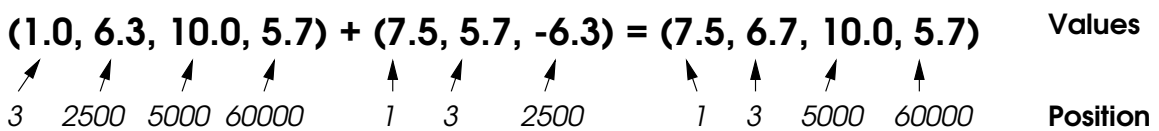
## Example

**Input:**

```
3 1.0 2500 6.3 5000 10.0 60000 5.7
1 7.5 3 5.7 2500 -6.3
10 0.0
15000 6.7
100 -1.0
100 1.0
-1
```

**Output:**

```
[1, 7.5], [3, 6.7], [5000, 10], [60000, 5.7]
[15000, 6.7]
empty vector
```



# 4 Line Drawing

A small company is developing a brand new computer system from scratch based on some brand new, but completely backwards-incompatible technology. They know that any new computer must support graphics, and since their technology is all new, they don't have access to the standard graphics packages. So, they came to you for your expert help.

A basic part of all graphics programs is to render graphical elements onto the screen, such as lines, rectangles, and polygons. It turns out that rectangles, polygons, and even circles can be drawn by connecting several short line segments, so all that is required at this point is to draw a line segment.

Since the hardware guys don't have the high-resolution mode on the monitor working yet, we only have access to text output - and so we will test the line drawing routine you are writing with text. Create an internal array of 50x20 characters to represent the screen, and clear all the characters to '.' to represent black. Then, your job is to take two points (x1,y1) - (x2,y2), and fill in the characters in your array that connect those points with 'X'. Then, print the array when you are finished to show that your routine works.

Note that this problem is slightly harder than it first appears because lines that are mostly horizontal or vertical should appear with all the X's inbetween the points filled in without any gaps. When you are computing which points to fill in, you may use floating point numbers internally. When you go to fill in the array, you must round your floats to the nearest integer.

## Input Format

The input consists of a number of line specifications, each on a separate line. Each line has 4 numbers that represents the two points of each line segment (x1 y1 x2 y2). All X coordinates are guaranteed to be in the range [0, 49] and all Y coordinates are guaranteed to be in the range [0, 19] The list of lines is terminated by a line containing only $-1$.

## Output Format

The output should be the 50x20 grid that represents the line connecting the two points with 'X's on a background of '.'s

## Examples

**Input 1:**

```
4 2 10 2
10 2 10 5
4 2 10 5
-1
```

**Output 1:**

```
................................................
................................................
....XXXXXXX.....................................
.....XX...X.....................................
.......XX.X.....................................
.........XX.....................................
................................................
................................................
................................................
................................................
................................................
................................................
................................................
................................................
................................................
................................................
................................................
```

**Input 2:**

```
0 0 1 19
1 19 3 0
3 0 6 19
6 19 10 0
10 0 15 19
15 19 21 0
21 0 28 19
28 19 36 0
36 0 45 19
-1
```

**Output 2:**

```
X..X......X..........X..............X.............
X..X......X..........X..............X.............
X..X......XX........X.X............X.X............
X..X.....X.X........X.X............X.X............
X..XX....X.X........X.X...........X...X...........
X.X.X....X.X.......X...X..........X...X...........
X.X.X....X..X......X...X.........X.....X..........
X.X.X....X..X......X....X........X.....X..........
X.X.X...X...X.....X.....X........X......X.........
X.X.X...X...X.....X.....X.......X.......X.........
.XX..X..X....X....X......X......X........X........
.XX..X..X....X....X......X.....X.........X........
.XX..X.X.....X...X.......X.....X..........X.......
.XX..X.X.....X...X........X....X..........X.......
.XX..X.X......X..X........X...X............X......
.X...X.X......X.X..........X..X............X......
.X....XX......X.X..........X.X..............X.....
.X....X.......X.X..........X.X..............X.....
.X....X........X............X................X....
.X....X........X............X................X....
```

# 5 Seating Diplomats

As the Secretary General of the United Nations, you are responsible for keeping everyone in the world relatively happy with each other and avoiding friction as much as possible. Since the United Nations has a lot of conferences which involve eating, you must seat all the ambassadors around a *circular* table without seating people in such a way that political friction might spring up. Your staff gives you a list of how many ambassadors there are and who must not be seated next to whom.

In order to make sure your decisions are foolproof, you write a program which you figure to be faultless to figure out the seating arrangements for you. Good luck with your conference and remember – the fate of the world is in your hands!

## Input Format

The input will first consist of a line containing the total number of diplomats, followed by lines containing pairs of diplomats who must NOT be seated together. The list will be terminated by a line containing only $-1$.

## Output Format

The output should be a sequence of numbers representing one possible correct seating of the diplomats (based on the input). The diplomats must be numbered from 1 to N (where N is the total number of diplomats given in the input). If no seating arrangement is possible, the program prints a 0 (zero).

## Examples

**Input 1:**

```
11
1 4
1 7
5 7
10 7
10 8
10 9
3 4
-1
```

**Output 1:**

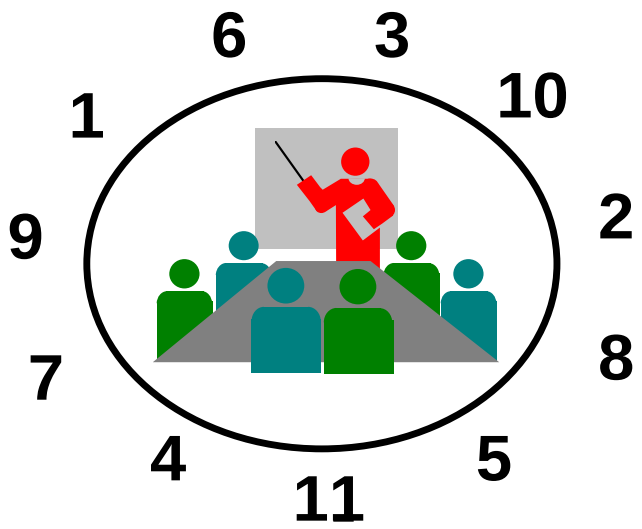
```
1 9 7 4 11 5 8 2 10 3 6
```

**Input 2:**

```
3
1 2
-1
```

**Output 2:**

```
0
```



# 6 Quaternion Calculator

Quaternions are a generalization of the complex number system, developed by Sir William Hamilton in the mid 19th century. Today quaternions are used in computer graphics and robotics, since a quaternion naturally encodes a rotation in 3-space, and multiplication of quaternions corresponds to composition of rotations. For our purposes, a quaternion is defined to be a quadruple of numbers

$$q = \begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{pmatrix},$$

where $q_i$ is of type `double`. Given two quaternions, $q$ and $p$, their sum, product, and negation are defined as follows:

$$q + p = \begin{pmatrix} q_0 + p_0 \\ q_1 + p_1 \\ q_2 + p_2 \\ q_3 + p_3 \end{pmatrix} \qquad q * p = \begin{pmatrix} q_0p_0 - q_1p_1 - q_2p_2 - q_3p_3 \\ q_0p_1 + q_1p_0 + q_2p_3 - q_3p_2 \\ q_0p_2 - q_1p_3 + q_2p_0 + q_3p_1 \\ q_0p_3 + q_1p_2 - q_2p_1 + q_3p_0 \end{pmatrix} \qquad -q = \begin{pmatrix} -q_0 \\ -q_1 \\ -q_2 \\ -q_3 \end{pmatrix}.$$

The objective of this problem is to design a simple calculator for quaternions. The calculator has 10 registers, numbered 0 through 9, each of which holds a single quaternion. Initially they are all set to zero. Your calculator will input a sequence of commands, each consisting of an operator followed by one or more arguments, and it will output the result of each operation.

## Input format

Each line will contain a single character "opcode" followed by one or more arguments, all separated by spaces. Here are the possible opcodes, and their action. The values $i$, $j$, and $k$ are integers in the range 0 through 9 (inclusive) and the values $q_i$ are of type double. You may assume that the input is in this format (e.g. you will not be given any illegal opcodes).

| Command line: | Meaning |
|---|---|
| `=` $q_0$ $q_1$ $q_2$ $q_3$ $i$ | Create quaternion $(q_0, q_1, q_2, q_3)$ and store in register $i$. |
| `+` $i$ $j$ $k$ | Add quaternions in registers $i$ and $j$ and store result in register $k$. |
| `*` $i$ $j$ $k$ | Multiply quaternions in registers $i$ and $j$ and store result in register $k$. |
| `-` $i$ $j$ | Negate the quaternion in register $i$ and store result in register $j$. |
| `Q` | Terminate the program. |

## Output format

With the exception of the 'Q' opcode, each operation that is executed outputs the result of the operation in the following format [$i$] ( $q_0$ $q_1$ $q_2$ $q_3$ ), where $i$ is the index of the register into which the result was stored.

## Examples

**Input 1:**

```
= 1 1 0 0  0
= 1 0 1 0  1
+ 0 1      2
* 0 1      9
- 1        9
Q
```

**Output 1:**

```
[0] ( 1 1 0 0 )
[1] ( 1 0 1 0 )
[2] ( 2 1 1 0 )
[9] ( 1 1 1 1 )
[9] ( -1 0 -1 0 )
```

**Input 2:**

```
= 1 .5 0 0  0
= 1 -1 0 0  1
= 0  0 1 0  2
= 0  0 0 1  3
- 3         3
+ 2 3       6
* 0 1       4
* 4 0       4
Q
```

**Output 2:**

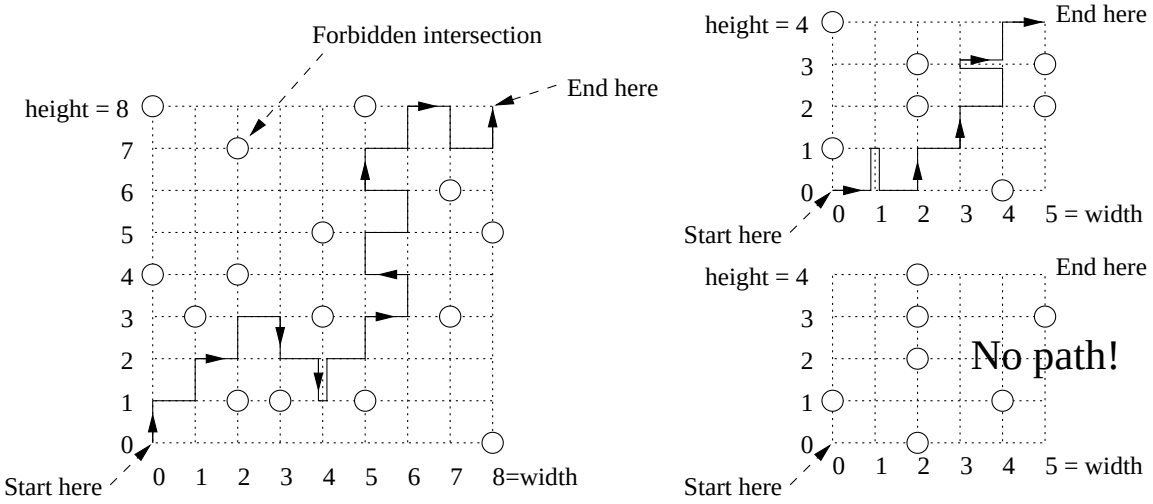
```
[0] ( 1 0.5 0 0 )
[1] ( 1 -1 0 0 )
[2] ( 0 0 1 0 )
[3] ( 0 0 0 1 )
[3] ( 0 0 0 -1 )
[6] ( 0 0 1 -1 )
[4] ( 1.5 -0.5 0 0 )
[4] ( 1.75 0.25 0 0 )
```

# 7 The Spy's Escape Route

This problem involves computing shortest path subject to motion constraints, as is common in robotics applications.

A spy is trying to find her way home from her secret hangout to her home base. The streets of the city are laid out on a square grid. Some intersections cannot be visited, because they have surveillance cameras. She can only change directions when she comes to an intersection. To avoid detection, she intentionally avoids walking straight whenever possible. For example, if she enters an intersection from its south side, then she must leave by going either east, west or back south again, but she cannot continue straight north. The spy is allowed to visit the same intersection more than once.

Write a program which determines whether the spy can make it home subject to these restrictions, and if so, outputs such a path. The figure below shows some possible situations. Notice that the path may backtrack at some points. In the situation in the lower right there is no path, because any path must go straight through the intersection at $(2, 1)$, and this is forbidden. The spy always starts in the lower left corner and goes to the upper right corner.




## Input format

The first line of input contains two integers, giving the width $w$ and height $h$ of the grid. There are $h + 1$ east-west streets with $y$-coordinates ranging from 0 to $h$, and there are $w + 1$ north-south streets, with coordinates ranging from 0 to $w$. The remaining lines contain the integer $x$ and $y$ coordinates of the forbidden intersections. They are not given in any particular order. The list is terminated by a line containing only $-1$. The spy starts her walk in the lower-left corner (with coordinates $(0, 0)$) and ends in the upper-right corner (with coordinates $(w, h)$). You may assume that $w$ and $h$ are each in the range from 0 to 100.

## Output format

If there is no path from the start to end or if any such path would require walking straight through an intersection, then output "no-path" on a single line. Otherwise if there is a path, output "path" on one line. Each of the remaining lines of output contains the $x$ and $y$ coordinates (separated by spaces) of the intersections along the path, starting with $(0,0)$ and ending with $(w,h)$. The path does not need to be the shorest possible.

## Example

The example below shows the input and output for the situations shown in the above figure on the right.

**Input 1:**

```
5 4
0 1
0 4
2 2
2 3
4 0
5 2
5 3
-1
```

**Output 1:**

```
path
0 0
1 0
1 1
1 0
2 0
2 1
3 1
3 2
4 2
4 3
3 3
4 3
4 4
5 4
```

**Input 2:**

```
5 4
0 1
2 0
2 2
2 3
2 4
4 1
5 3
-1
```

**Output 2:**

```
no-path
```

## 8 Comparing Trees

You work on a submarine and have just received an updated version the submarine's manual, which is several gigabytes in size. The captain doesn't want the crew to waste time leafing through the new version trying to figure out how it differs from the one they read in training. On the other hand, she would hate to accidentally launch a missile when trying to turn on the microwave. As the grunt on board, you're assigned the task of comparing the old and new versions and marking how they differ from each other. For this task, the captain graciously provides you with a highlighter and lots of coffee.

You have a better idea. Since both versions are available in electronic form, you think of writing a program that would compare them automatically. The manual is organized hierarchically in volumes, chapters, sections, subsections, paragraphs, etc., so you model it using a tree (described below). All you need to do now is write a program that takes two trees and outputs a description of how they differ.

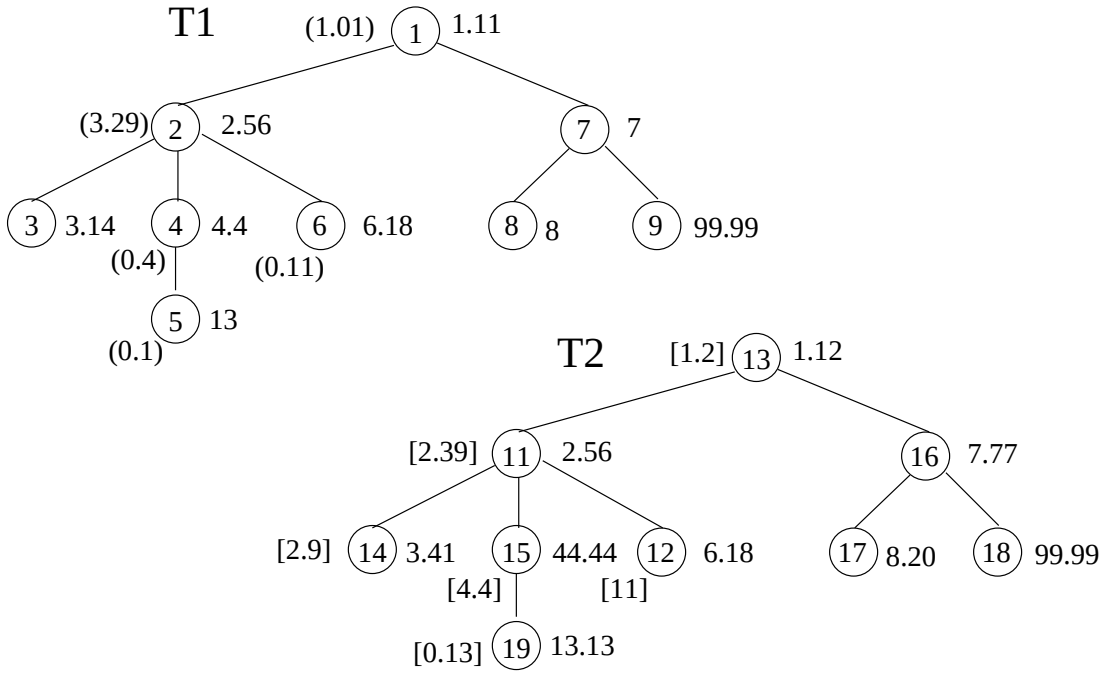



Figure 1: Comparing Trees

### Trees

We use the term *tree* to mean a *rooted, ordered, labeled tree*, such as those depicted in Figure 1. (Computer scientists like to draw their trees upside-down.) Each tree has a unique "top" node called its *root*. In the figure, the node numbered 1 is the root of tree $T_1$. (The node number is used to identify a node and is indicated within the circle representing the node.) Each node in a tree also has a *label*, which is indicated next to that node in the figure. For this problem, we assume that labels are real numbers. For example, the label of node 4 is 4.40. Each node except the root also has a unique *parent* node. In the figure, we connect a node to its parent (depicted above the

node) using a line. For example, the parent of node 7 is node 1. A node is called it's parent's *child*. Nodes that do not have any children are called *leaf nodes*; the rest are called *interior nodes*. The children of each interior node are ordered, and the figure depicts the children of each node left-to-right in this order. For example, the children of node 2, in order, are 3, 4, and 6.

All the nodes that lie on the path (of length zero or more) from a node to the root are called that node's *ancestors*. Note that every node is its own ancestor. If a node $n_1$ is the ancestor of node $n_2$, then $n_2$ is called a *descendant* of $n_1$. The tree consisting of all the descendants of of a node $n$ is called its *subtree*. The length of the path from a node to the root is called that node's *depth*. Thus, the depth of the root is always 0. In our example, the depth of node 4 is 2.

The *preorder list* of a tree consists of the root followed by the preorder list of the subtrees rooted at its children, in order. For example, the preorder list of tree $T_1$ is $(1, 2, \ldots, 9)$. The preorder list of tree $T_2$ is (13, 11, 14, 15, 19, 12, 16, 17, 18).

## Editing Trees

Given such a tree, we can *edit* (modify) it using the following three kinds of *edit operations*:

- An *update operation* can be used to change the label of a node. For example, we can change the label of node 3 to 3.41 by using the operation *upd*$(3, 3.41)$. That is, the operation *upd*$(n, l)$ changes the label of node $n$ to $l$.
- A *delete operation* can be used to remove any subtree from a given tree. For example, we can remove the subtree rooted at node 4 by using the operation *del*$(4)$. That is, the operation *del*$(n)$ applied to any tree containing $n$ all of $n$'s descendants (including $n$) from the tree.
- An *insert operation* can be used to attach one tree at some point in another tree. Refer to Figure 2. The third arrow in Figure 2 depicts the tree $T'$ being attached as the second child of the node 2 in tree $T_3$ by using the operation *ins*$(T', 5, 2)$. That is, an operation *ins*$(T, n, i)$ makes $T$ the $i$th child of node $n$.

An *edit script* is a sequence of such edit operations. An edit script is applied to a tree by applying the operations one after another in the order they are listed. Figure 2 depicts the edit script (*upd*$(1, 1.12)$, *upd*$(3, 3.41)$, *del*$(4)$, *ins*$(T', 2, 2)$, *upd*$(7, 7.77)$, *upd*$(8, 8.20)$) applied to tree $T_1$ from Figure 1. Note that the resulting tree is identical to the tree $T_2$ (except for node identifiers). We say that our *edit script transforms* $T_1$ to $T_2$.

## Costs

Each edit operation has a *cost* associated with it as described below:

- The *cost of an update operation* is simply the absolute value of the difference between the old and new labels. Thus updating a label 1.11 to 1.12 costs 0.01 units and updating a label 9.99 to 99.9 costs 89.91 units.
- Each node has a *deletion cost* that depends on its label and that is specified as part of the input. In Figure 1, the deletion cost of a node is indicated in parentheses next to the node. If no such number appears next to a node, its deletion cost is 1 unit by default. The *cost of a delete operation* is the sum of the costs of deleting all the nodes in the deleted subtree. For example, the operation *del*$(4)$ in our example costs $0.4 + 0.1 = 0.5$ units.

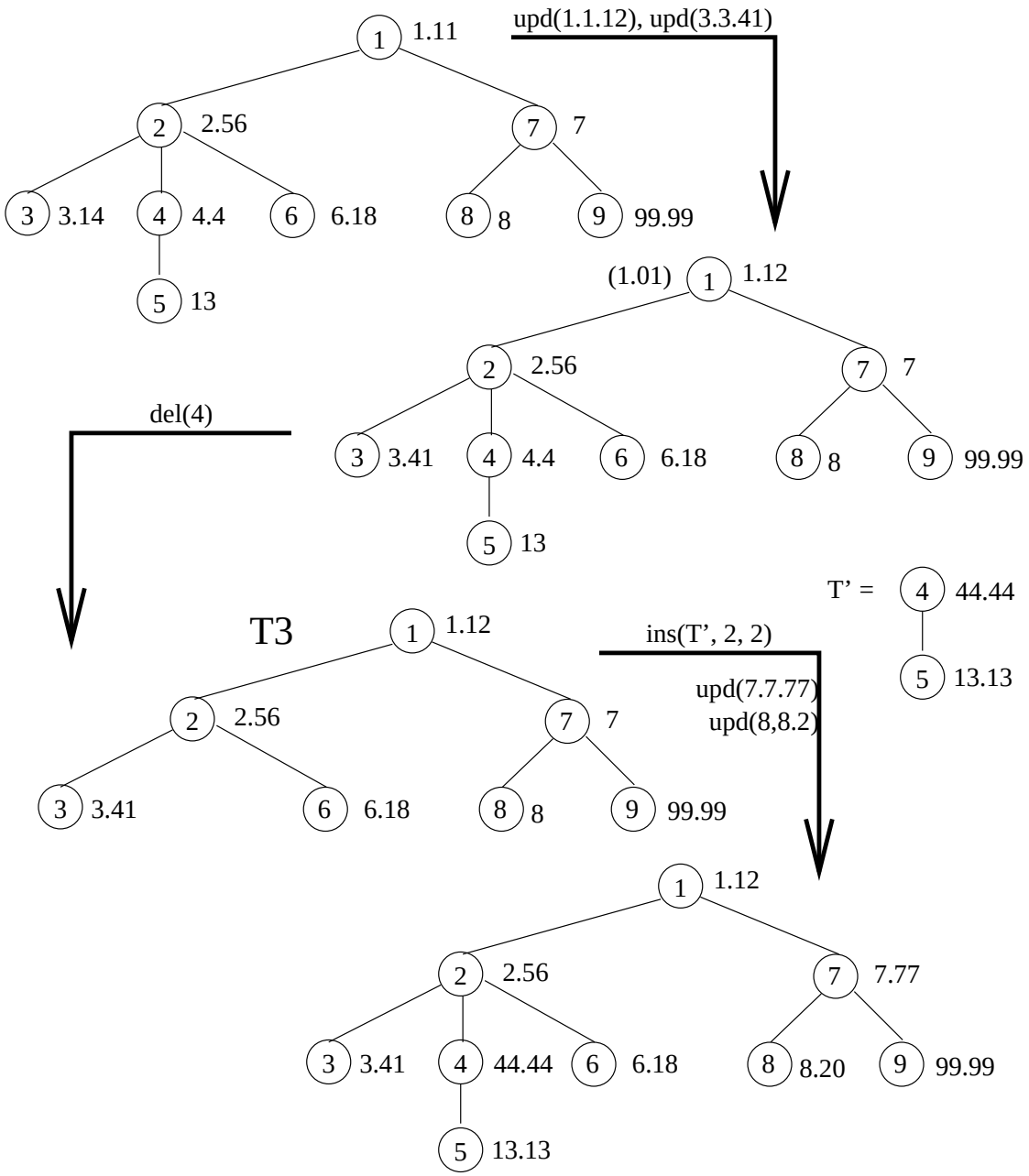



Figure 2: Transforming a tree using an edit script

- Similarly, each node has an *insertion cost* that depends on its label and that is specified as part of the input. In the figure, the insertion cost of a node is indicated in square brackets next to the node. If no such number appears next to a node, its insertion cost is 1 unit by default. The *cost of an insert operation* is the sum of the insertion costs all the nodes in the inserted subtree. For example, using the costs indicated in Figure 1, the cost of the operation $ins(T', 2, 2)$ in earlier example is $4.4 + 0.13 = 0.53$ units.

  The cost of inserting or deleting a node whose label does not appear in any of the input trees is 1 unit by default.

The *cost of an edit script* is the sum of the costs of the operations it contains.

## Goal

Your goal is to write a program that takes as input two trees (and the node insertion and deletion costs) and produces as output the cost of a minimum-cost edit script that transforms one tree to another. (You need not compute such a minimum-cost edit script; you only need its cost.) As was the case for the trees in Figure 1, there is no correspondence between the node identifiers in the two trees; thus they are not specified in the input. You are also required to pretty-print the input trees as described below.

## Input format

The input file first lists the nodes in the first tree in preorder, one per line. For each node, the input file contains a line with three numbers that are separated using spaces. The first number is the depth of the node. The second is that node's label. The third is that node's deletion cost. After all the nodes in the first tree are listed in this manner, the input file contains a line "`-1 0 0`" to denote the end of the tree's listing.

Next, the input contains a blank line followed by a listing of the second tree. The second tree is listed using the method described above, the only difference being that the third number on each line is now the node's insertion cost. This listing of the nodes in the second tree is also followed by a line "`-1 0 0`" to denote the end of the listing.

Assume that all real numbers in the input are in the range $[0, 99.99]$ (inclusive).

## Output format

The output consists of two parts. First, you pretty-print the two input trees. Second, you report the cost of a minimum-cost edit script that transforms the first tree to the second.

You must first output the first tree by listing one node per line, in preorder. Each such line must begin with $4d$ space characters, where $d$ is the depth of the node. Next, it contains the node's label, with two digits before and two digits after the decimal point. If the label is less than 10, include a leading space character. Thus, 9.87 is output as "␣`9.87`" where ␣ denotes the ASCII space character. This label is followed by a single space character, followed by the node's deletion cost listed in the same format as that used for the label.

Next, you must output a blank line, followed by the listing of the second tree in the above format (with the deletion cost replaced by the insertion cost).

Finally, you output a blank line followed by a line containing the string "`Distance:`" followed by a space character followed by the the cost of the minimum-cost edit script. For outputting this

cost, use the same format you used for the tree labels. (You can assume that the cost will always be a real number in the range $[0, 99.99]$.)

Do not include anything else in your output, not even blank lines.

## Example

For the trees depicted in Figure 1, the input and output is shown below. (For clarity, we use the ␣ character to denote the ASCII space character.) Note that the space characters at the beginning of lines in the input are optional; your program should work whether or not they occur in the input. As it turns out, the edit script depicted in Figure 2 is a minimum-cost edit script for this input, and its cost is 6.28, as reported in the output.

**Input 1:**

```
0␣1.11␣1.01
␣␣1␣2.56␣3.29
␣␣␣␣2␣3.14␣1
␣␣␣␣2␣4.4␣0.4
␣␣␣␣␣␣␣3␣13␣0.1
␣␣␣␣2␣6.18␣0.11
␣␣1␣7␣1
␣␣␣␣2␣8␣1
␣␣␣␣2␣99.99␣1
-1␣0␣0
```

```
0␣1.12␣1.20
␣␣1␣2.56␣2.39
␣␣␣␣2␣3.41␣2.9
␣␣␣␣2␣44.44␣4.4
␣␣␣␣␣␣␣3␣13.13␣0.13
␣␣␣␣2␣6.18␣11.0
␣␣1␣7.77␣1
␣␣␣␣2␣8.2␣1
␣␣␣␣2␣99.99␣1
-1␣0␣0
```

**Output 1:**

```
␣1.11␣␣1.01
␣␣␣␣␣␣2.56␣␣3.29
␣␣␣␣␣␣␣␣␣␣3.14␣␣1.00
␣␣␣␣␣␣␣␣␣␣4.40␣␣0.40
␣␣␣␣␣␣␣␣␣␣␣␣␣13.00␣␣0.10
␣␣␣␣␣␣␣␣␣␣6.18␣␣0.11
␣␣␣␣␣␣7.00␣␣1.00
␣␣␣␣␣␣␣␣␣␣8.00␣␣1.00
␣␣␣␣␣␣␣␣␣99.99␣␣1.00

␣1.12␣␣1.20
␣␣␣␣␣2.56␣␣2.39
␣␣␣␣␣␣␣␣␣␣3.41␣␣2.90
␣␣␣␣␣␣␣␣␣44.44␣␣4.40
␣␣␣␣␣␣␣␣␣␣␣␣␣13.13␣␣0.13
␣␣␣␣␣␣␣␣␣␣6.18␣11.00
␣␣␣␣␣␣␣␣␣␣␣␣␣13.13␣␣0.13
␣␣␣␣␣␣␣␣␣␣6.18␣11.00
␣␣␣␣␣␣7.77␣␣1.00
␣␣␣␣␣␣␣␣␣␣8.20␣␣1.00
␣␣␣␣␣␣␣␣␣99.99␣␣1.00

Distance:␣␣6.28
```

**Input 2:**

```
0␣1␣1
␣␣1␣3␣1
␣␣1␣2.2␣1.1
␣␣␣␣2␣3.14␣1.73
␣␣␣␣2␣7.8␣0.5
␣␣1␣1␣1
-1␣0␣0
```

```
0␣1␣1
␣␣1␣3␣1
␣␣1␣2␣1.1
␣␣␣␣2␣3.14␣1.73
␣␣␣␣2␣8.7␣0.3
␣␣1␣1␣1
-1␣0␣0
```

**Output 2:**

```
␣1.00␣␣1.00
␣␣␣␣␣␣3.00␣␣1.00
␣␣␣␣␣␣2.20␣␣1.10
␣␣␣␣␣␣␣␣␣␣3.14␣␣1.73
␣␣␣␣␣␣␣␣␣␣7.80␣␣0.50
␣␣␣␣␣␣1.00␣␣1.00

␣1.00␣␣1.00
␣␣␣␣␣␣3.00␣␣1.00
␣␣␣␣␣␣2.00␣␣1.10
␣␣␣␣␣␣␣␣␣␣3.14␣␣1.73
␣␣␣␣␣␣␣␣␣␣8.70␣␣0.30
␣␣␣␣␣␣1.00␣␣1.00

Distance:␣␣1.00
```