Colombian Collegiate Programming League
CCPL 2013

Contest 9 -- August 24

Problems

This set contains 9 problems; pages 1 to 18.

(Borrowed from several sources online.)

Official site `http://programmingleague.org`

Official Twitter account @CCPL2003

A - Apply A Cold Compress

Source file name: `compress.c`, `compress.cpp` *or* `compress.java`

Many techniques for compressing digital graphics focus on identifying and describing regions of a single uniform character. Here is a simple technique for compressing black-and-white images (which could be easily extended to color). The basic idea is to repeatedly split the original picture in half, either vertically or horizontally, until each of the resulting sub-pictures contains only a single color.

A rectangular digital graphic is described by a ''compression-expression,'' defined as follows: Each compression-expression begins with a two-bit tag, which may be followed by additional compression-expressions depending upon the tag value. The tag values are interpreted as follows:

00 A square region that consists entirely of black pixels. This region may be a single pixel, a 2×2 square, a 3×3 square, etc., depending upon context.

11 A square region that consists entirely of white pixels. This region may be a single pixel, a 2×2 square, a 3×3 square, etc., depending upon context.

10 A horizontal split. This is followed by two compression expressions. The picture produced by a split is formed by taking the pictures denoted by each of those two expressions and placing them along-side one another, the first picture to the left and the second to the right. Horizontal splits are only possible between two pictures of the same height.

01 A vertical split. This is followed by two compression expressions. The picture produced by a split is formed by taking the pictures denoted by each of those two expressions and placing them along-size one another, the first picture on the top and the second underneath it. Vertical splits are only possible between two pictures of the same width.

When interpreting splits, it may be necessary to change the scale of the components to make them compatible. For example, given a 2 : 6 picture A (i.e., 2 pixels wide, 6 pixels high) and a 3 : 4 picture B:

- A vertical split involving these two is possible only if we scale A by a factor of 3, making it 6 : 18, and scale B by a factor of 2, making it 6 : 8. The resulting combined picture would have size 6 : 26.
- A horizontal split involving these two is possible only if we scale A by a factor of 2, making it 4 : 12, and scale B by a factor of 3, making it 9 : 12. The resulting combined picture would have size 13 : 12.

For example, using '`X`' and '` `' to denote black and white pixels, respectively, the expression 00 denotes the picture

```
---
|X|
---
```

and the expression 1000010011 denotes

```
-----
|XXX|
|XX |
-----
```

Examination of this format will show that for any given compression-expression, there is some smallest picture that can be denoted by that expression, but the same expression can also denote pictures twice the size of the smallest one, three times the size, etc.

Input

Each line of the input will contain a compression-expression, presented as a single line containing an arbitrary number of 0's and 1's. The input ends following the line with the final compression-expression.

All input sets used in this problem will be valid compression-expressions.

The input must be read from standard input.

Output

For each line of input, your should print the smallest black-and-white picture denoted by that expression, drawn in '`X`' (black) and ' ' (white), as above, and framed in '`-`' and '`|`' characters as shown in the examples. There should be no characters or whitespace outside your frame except for the newlines terminating each line.

There should be no blank lines in your output.

The output must be written to standard output.

Sample input

```
00
100010111001001011101111011110001000111
```

Output for the sample input

```
---
|X|
---
----------------
|XXXX    XXX   |
|XXXX    XXX   |
|XXXX    XXX   |
|XXXX       XX |
----------------
```

B - Spelling Be

Source file name: `spelling.c, spelling.cpp` *or* `spelling.java`

It's a simple requirement your company has, really --every document should be spell-checked before it's sent out to a customer. Unfortunately, while word processing documents are easily spell-checked, your employees have not been checking email every time they send out a message. So you've come up with a little improvement. You are going to write a program that will check email on its way out. You will spell-check each message, and if you find any spelling errors, it will be returned to the sender for correction.

When you announced this plan, one of your coworkers fell off their chair laughing, saying that you couldn't possibly anticipate every name, technical acronym, and other terms that might appear in an email. Undaunted, however, you are going to test-run your code with an online dictionary and some sample emails you have collected.

Input

The input consists of two sections, the dictionary and the emails. The first line of input specifies the number of words in the dictionary, followed by that many lines, with one word per line. There is no whitespace before, after, or in any words, although there may be apostrophes or hyphens in the words, which are considered part of the word (i.e., ''bobs'' is different than ''bob's''). There will be no duplicate words. All words will be in lower case.

Following that are the emails. The first line of this section has the number of emails in the input. Following that line begins the first email. It has been pre-processed, so it consists of one word per line, with no punctuation (other than apostrophes and hyphens) or whitespace, and all words are in lower case. The last word in each email is followed by a line with only -1. Each email will have at least one word.

The input must be read from standard input.

Output

For each email, you must either print:

```
Email X is spelled correctly.
```

where *X* begins with 1 and counts up. Or, if a word is found that is not in the dictionary, print out:

```
Email X is not spelled correctly.
```

followed by a list of unknown words in the order that you find them, one per line. If an unknown word is found multiple times, it should be printed multiple times.

There are no spaces between datasets. Following the output for the final dataset, print a line stating “`End of Output`”.

The output must be written to standard output.

Sample input	**Output for the sample input**

`6`
`alice`
`am`
`bitterly`
`i`
`leaving`
`you`
`1`
`dear`
`bob`
`i`
`am`
`leaving`
`you`
`bitterly`
`alice`
`-1`
	`Email 1 is not spelled correctly.`
	`dear`
	`bob`
	`End of Output`

C - Chambers Ceramic Conundrum

Source file name: `chambers.c, chambers.cpp` *or* `chambers.java`

The Chambers Construction Company has a major contract to deliver a tile floor on schedule for its largest customer (Moneybags to Spare Inc). Unfortunately, the clerk who ordered tiles won the lottery just after ordering the tiles for this room, and did not write down where to place each tile to make the room fit. Normally this would not be a problem, except the tiles that were ordered are not all squares. They are each made up of 4 square segments, but will take on all possible shapes shown here:

```
XXXX      XX      XX       XX      X        XXX     X
          XX       XX     XX       XXX      X      XXX
```

Given that the project is under an extremely tight schedule, it is not possible to reorder the tiles in a more standard manner. Instead when the 9 tiles come in, you will need to figure out how to place the tiles (or if there is no way to set the pieces correctly). The tiles in the box are ordered from *A* to *I*. The room that needs to be tiled is 6 segments on each side.

Placing Tiles. A middle-manager at CCC has come up with an algorithm that they will give to the tiler to tile the room. The tiler has come to you to write a program to determine what pattern will be successful without having to try all of them with the physical tiles. The tiler will always start with the top left corner of the room. After placing the first tile, they will work their way from left to right and from top to bottom, placing the next tile such that it will fill the leftmost open space on the top line with an open space.

For instance, if the layout currently had the following tiles (*A* and *B*) placed:

```
AABBBZ
 AA B
```

The next tile to place would be placed such that position *Z* is filled.

To make matters easier, the tiler will always place the earliest tile in the box that could successfully fill that position. For instance, if either tile *B* or *C* could fill that position, the tiler will choose *B*. Furthermore, they will always place the tile as they are oriented above if possible. They will then attempt to rotate the tile 90 degrees clockwise and place it (possibly doing this 3 times).

NOTE: Remember the tiles may be rotated, but they may not be flipped.

A tile may not extend outside the 6×6 room, or overlap with another tile. A room is considered successfully tiled if the given set of tiles completely tile the room using the above algorithm.

The tiler will continue laying tiles according to this algorithm until finishing the room, or discovering that the room cannot be finished using the previous choices. If the room cannot be finished, the tiler will backtrack, considering the remaining rotations of the previous tile, and then the remaining tile. The tiler will continue to backtrack, one tile at a time, until all combinations have been tried or the room cannot be successfully tiled.

Input

The first line of input will indicate the number *N* of data sets.

The next *N* lines will each contain 9 numbers. These numbers indicate the shape of each tile. The first entry on the line will correspond to tile *A*, the second to tile *B*, ..., the 9th for tile *I*. Each number will reflect the layout of a tile as specified above (the left most layout is 1, the rightmost is 7).

The input must be read from standard input.

Output

For each data set, the first line of output should indicate the index of the data set, starting with 1. (''Data Set 1'')

The next line of input will indicate the floor may or may not be tiled successfully. (''The floor may be tiled.'' or ''The floor may not be tiled.'').

The next 6 lines would then display the graph of floor illustrating the final layout, if possible. To make it easy to understand, each tile set should be marked *A*-*I* corresponding to the order they were in the input line. Each line will have exactly 6 characters indicating the tile segment, followed by a newline.

A blank line should appear after each data set.

The line stating ''End of Output'' should appear after the last data set.

The output must be written to standard output.

Sample input

```
2
2 2 2 2 2 2 2 2 3
1 1 2 1 1 2 1 1 2
```

Output for the sample input

```
Data Set 1
The floor may not be tiled.

Data Set 2
The floor may be tiled.
AAAABD
CCEGBD
CCEGBD
FFEGBD
FFEGII
HHHHII

End of Output
```

D - In Defence of a Garden

Source file name: `garden.c, garden.cpp` *or* `garden.java`

It's said that ''Necessity is the mother of invention,'' but some people think that ''Laziness'' is a more likely parent.

Hubert Greenthumb hated digging fence posts. But he knew that, without a fence around his garden, deer from the nearby woods would eat his vegetables before he could harvest them.

Being something of a tinkerer, he retired to his workshop with a small garden tractor, some out-of-date computer chips, and a couple of robot arms he had picked up at a bankruptcy auction from a failed ''.com'' high-tech company. After two days of work, he emerged as the proud inventor of the Greenthumb Automatic Garden Fence Layer (pat. pending).

To his skeptical wife (who observed that he could easily have built the fence in half the time it took to construct this machine), he explained that he needed only to program in the desired fence shape, and the machine would proceed to chug around the yard, laying down a fence in 1-foot sections until the job had been completed.

Hubert proceeded to key in instructions to enclose a square area, 25 feet on a side, of his 100 by 100 yard. He set the machine to operating and went inside for a celebratory drink.

When he emerged, he discovered that the machine had laid down fence in an elaborate, possibly random walk about his lawn. Unwilling to actually admit that anything had gone wrong, he announced his intention to plant within the garden actually laid out by the machine, as if he had wanted it that way all along. Any section of the yard that was no longer accessible to the deer (enclosed by the fence) would be considered as garden space.

''Fine,'' sighed his wife, ''but we'll need to know just how many square feet of garden we have so that we can buy an appropriate amount of seeds.'' Hubert gamely began to trace out the fence laid down by the machine. ''Let's see, it went North for 5 feet, then West for 3 feet, ...''

Note: Hubert's yard can be divided into a grid of 100×100 feet, with each grid box being 1 foot by 1 foot. The robot moves along the edges of the boxes. As the robot moves, it builds a fence from vertex to vertex of the grid (intersections of the lines).

Note: Because the robot moves along the edges of the grid, you can ignore the amount of space the fence occupies. For example, if the robot moves North one, East one, South one, and West one, it has enclosed one square foot of garden space.

Input

The first line of the input will contain the number of data sets. There are no blank lines before or after each data set.

The first line of each data set will contain three integers (X Y Z), indicating the X and Y position of the starting point on the grid, and the number of moves the robot makes. (X, Y, and Z are all non-negative integers; X is the number of feet from the western edge; Y is the number of feet from the southern edge of the yard), with $0 \leq X, Y \leq 100$.

The next Z lines will contain a character D and an integer F, separated by a space. The character D indicates the direction (N, S, E, W) and the integer F indicates how far in that direction the robot traveled.

The path will never leave the 100×100 yard. The path may or may not be closed. It may cross itself or retrace its steps (walk along lines in the grid it previously laid fence). It automatically stops building fence until it moves onto an edge without fence on it.

The input must be read from standard input.

Output

For each data set, output a single line of the form

```
Data Set N: Q square feet.
```

where N is the data set number (from 1) and Q is the number of square feet that are enclosed so they may be used for the garden.

After the last line of output, print ''`End of Output`'' on a line by itself.

The output must be written to standard output.

Sample input	**Output for the sample input**

`1`
`0 0 8`
`N 25`
`E 25`
`N 25`
`E 25`
`S 25`
`W 25`
`S 25`
`W 25`
	`Data Set 1: 1250 square feet.`
	`End of Output`

E - Component Testing

Source file name: `component.c, component.cpp` *or* `component.java`

The engineers at ACM Corp. have just developed some new components. They plan to spend the next two months thoroughly reviewing and testing these new components. The components are categorized into several different classes, depending on their complexity and importance. Components in different classes may require different number of reviewers, whereas components in the same class always require the same number of reviewers.

There are also several different job titles at ACM Corp. Each engineer holds a single job title. All engineers holding a given job title have the same limit on the number of components that they can review. Note that an engineer can be assigned to review any collection of components and will be able to complete the task, regardless of which classes the components belong to. An engineer may review some components of the same class, and others from different classes, but an engineer cannot review the same component more than once.

Can the engineers complete their goal and finish testing all components in two months?

Input

There will be multiple test cases in the input.

The first line of each test case contains two integers n ($1 \leq n \leq 10,000$) and m ($1 \leq m \leq 10,000$), where n is the number of component classes and m is the number of engineer job titles.

Each of the next n lines contains two integers j ($1 \leq j \leq 100,000$) and c ($0 \leq c \leq 100,000$), indicating that there are j components in this class and that each component requires at least c different reviewers.

Then each of the next m lines each contains two integers k ($1 \leq k \leq 100,000$) and d ($0 \leq d \leq 100,000$), indicating that there are k engineers with this job title and that each engineer may be assigned to review at most d components.

The input will end with a line with two 0s.

The input must be read from standard input.

Output

For each test case, print a single line containing 1 if it is possible for the engineers to finish testing all of the components and 0 otherwise.

The output must be written to standard output.

Sample input	**Output for the sample input**

`3 2`
`2 3`
`1 2`
`2 1`
`2 2`
`2 3`
`5 2`
`1 1`
`1 3`
`1 1`
`1 3`
`1 1`
`1 20`
`1 4`
`0 0`
	`1`
	`0`

F - Funhouse

Source file name: `funhouse.c, funhouse.cpp` *or* `funhouse.java`

An amusement park is building a new walk-through funhouse. It is being built in a large space: 1000*ft* × 1000*ft*. The park will build walls in the space, separating it into rooms. Some walls will have doors so that guests can move between rooms. Guests will enter through specially marked entrances, and exit through specially marked exits. They can move through the space as they wish -in fact, there may be many different ways of moving from the entrance to the exit. Of course, there will be many amusing things along the way.

The park designers want to install *shakerboards*, which are moving floors, to surprise the guests. To enhance the surprise factor, wherever they install shakerboards, they'll fill the whole room with them. That way, the boards won't stand out.

The designers want every guest in the funhouse to experience shakerboards, but, as you can imagine, shakerboards are expensive, so the park wants to cover as little space with them as possible.

Given a description of a funhouse design, what's the smallest area that must be covered with shakerboards to assure that every guest experiences them?

Input

There will be several data sets. Each data set will begin with a line with one integer n ($3 \leq n \leq 1000$), which is the number of walls.

Each of the next n lines will describe a wall, like this:

$$x_1\ y_1\ x_2\ y_2\ EXDW$$

where (x_1, y_1) and (x_2, y_2) are the endpoints of the wall, and $EXDW$ is a single capital letter: 'E' for an entrance, 'X' for an exit, 'D' for an interior wall with a door, and 'W' for any wall without a door. 'E' and 'X' are guaranteed to only appear on exterior walls, and 'D' is guaranteed to only appear on interior walls. 'W' may appear on either.

The endpoint coordinates will be integers, with values between 0 and 1000 inclusive. Walls will never intersect each other in any way or be coincident, except for sharing endpoints. Every endpoint will be coincident with another wall's endpoint. No wall will have zero length. There is guaranteed to be at least one way to get from every entrance to some exit and to every exit from some entrance. The funhouse will consist of a single building. In order to provide power throughout the building, every interior wall is connected to an exterior wall either directly or indirectly via a series of other walls.

End of input will be marked by a line with a single 0.

The input must be read from standard input.

Output

For each test case, print a single line containing the smallest area that the park owners must cover with shakerboards so that every guest in the funhouse will experience them. This should be printed as a floating point number to one decimal digit precision.

The output must be written to standard output.

Sample input	**Output for the sample input**

`6`
`0 0 100 0 W`
`0 0 0 100 E`
`0 100 100 100 W`
`100 0 100 100 D`
`100 0 200 0 W`
`200 0 100 100 X`
`14`
`0 0 100 0 W`
`100 0 110 0 E`
`110 0 190 0 W`
`190 0 200 0 E`
`0 0 0 100 W`
`100 0 100 100 D`
`200 0 200 100 W`
`0 100 100 100 D`
`100 100 200 100 D`
`0 100 0 150 X`
`100 100 100 150 D`
`200 100 200 150 X`
`0 150 100 150 W`
`100 150 200 150 W`
`0`
	`5000.0`
	`10000.0`

G - A Terribly Grimm Problem

Source file name: `grimm.c, grimm.cpp` *or* `grimm.java`

Grimm's conjecture states that to each element of a set of consecutive composite numbers one can assign a distinct prime that divides it.

For example, for the range 242 to 250, one can assign distinct primes as follows:

242	243	244	245	246	247	248	249	250
2	3	61	7	41	13	31	83	5

Given the lower and upper bounds of a sequence of composite numbers, find a distinct prime for each. If there is more than one such assignment, output the one with the smallest first prime. If there is still more than one, output the one with the smallest second prime, and so on.

Input

There may be several data sets.

Each data set will consist of a single line with two integers, L and H ($4 \leq L < H \leq 10^{10}$). It is guaranteed that all the numbers in the range from $L \ldots H$, inclusive, are composite.

The input will end with a line with two 0s.

The input must be read from standard input.

Output

For each data set, print a single line containing the set of unique primes, in order, separated by a single space.

The output must be written to standard output.

Sample input	**Output for the sample input**

`242 250`
`8 10`
`0 0`
	`2 3 61 7 41 13 31 83 5`
	`2 3 5`

H - Cantoring Along

Source file name: `cantor.c`, `cantor.cpp` *or* `cantor.java`

The Cantor set was discovered by Georg Cantor. It is one of the simpler fractals. It is the result of an infinite process, so for this program, printing an approximation of the whole set is enough.

The following steps describe one way of obtaining the desired output for a given order Cantor set:

1. Start with a string of dashes, with length 3^{order}.
2. Replace the middle third of the line of dashes with spaces. You are left with two lines of dashes at each end of the original string.
3. Replace the middle third of each line of dashes with spaces. Repeat until the lines consist of a single dash.

For example, if the order of approximation is 3, start with a string of 27 dashes:

```
---------------------------
```

Remove the middle third of the string:

```
---------         ---------
```

and remove the middle third of each piece:

```
---   ---         ---   ---
```

and again:

```
- -   - -         - -   - -
```

The process stops here, when the groups of dashes are all of length 1. You should not print the intermediate steps in your program. Only the final result, given by the last line above, should be displayed.

Input

Each line of input will be a single number between 0 and 12, inclusive, indicating the order of the approximation. The input stops when the end-of-file is reached.

The input must be read from standard input.

Output

You must output the approximation of the Cantor set, followed by a newline. There is no whitespace before or after your Cantor set approximation. The only characters that should appear on your line are '-' and ' '. Each set is followed by a newline, but there should be no extra newlines in your output.

The output must be written to standard output.

Sample input	**Output for the sample input**
`0`	`-`
`1`	`- -`
`3`	`- -   - -         - -   - -`
`2`	`- -   - -`

I - A Simple Question of Chemistry

Source file name: `chemistry.c, chemistry.cpp` *or* `chemistry.java`

Your chemistry lab instructor is a very enthusiastic graduate student who clearly has forgotten what their undergraduate Chemistry 101 lab experience was like. Your instructor has come up with the brilliant idea that you will monitor the temperature of your mixture every minute for the entire lab. You will then plot the rate of change for the entire duration of the lab.

Being a promising computer scientist, you know you can automate part of this procedure, so you are writing a program you can run on your laptop during chemistry labs. (Laptops are only occasionally dissolved by the chemicals used in such labs.) You will write a program that will let you enter in each temperature as you observe it. The program will then calculate the difference between this temperature and the previous one, and print out the difference. Then you can feed this input into a simple graphing program and finish your plot before you leave the chemistry lab.

Input

The input is a series of temperatures, one per line, ranging from -10 to 200. The temperatures may be specified up to two decimal places. After the final observation, the number 999 will indicate the end of the input data stream. All data sets will have at least two temperature observations.

The input must be read from standard input.

Output

Your program should output a series of differences between each temperature and the previous temperature. There is one fewer difference observed than the number of temperature observations (output nothing for the first temperature). Differences are always output to two decimal points, with no leading zeroes (except for the ones place for a number less than 1, such as 0.01) or spaces.

After the final output, print a line with ''`End of Output`''.

The output must be written to standard output.

Sample input	**Output for the sample input**

`10.0`
`12.05`
`30.25`
`20`
`999`
	`2.05`
	`18.20`
	`-10.25`
	`End of Output`