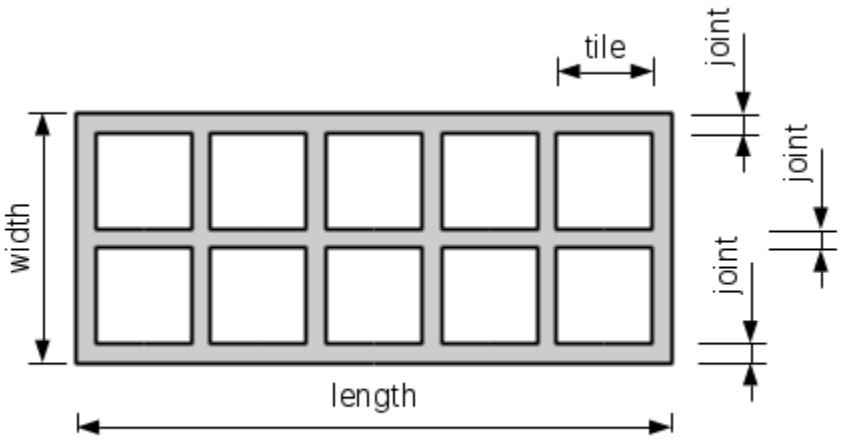
The task is to develop a program that finds the dimensions of a bathroom.

A bathroom will be of a rectangular (or square) top view, however, the dimensions need to be computed. We are limited by the minimum width and length, i.e. the actual width and length must be greater or equal to the given minima. Next, the bathroom will be tiled by tiles, the size of the tiles may vary. The dimensions of the bathroom must be set such that any given tile size fits the dimensions (of course, we must not cut the tiles).

The input of the program are decimals, each decimal represents some dimension in centimeters with the precision of one tenth (thus the efficient precision is 1mm). First, the program is given the minimum bathroom width and length. Next, there is a list of tiles, each tiling is described by two decimals. The first decimal is the dimension of the tile (the tiles are squares), the second decimal is the size of the joint (a "gap" between the tiles). The list of tiles ends with the end-of-file (EOF) condition in the standard input.

The output of the program is the smallest dimension of the bathroom such that the computed dimension is greater or equal to the requested minimum and the output dimension could be tiled by any of the tiles in the input without the need to cut the tiles.



The program must validate input data. If the input is invalid, the program must detect it, it shall output an error message (see below) and terminate. If displayed, the error message must be displayed to the standard output (do not send it to the error output) and the error message must be terminated by a newline (\n). The input is considered invalid, if:

- the input dimensions are not valid decimals,
- any input dimension is negative,
- any input dimension is given with precision greater than 1mm,
- any input dimension exceeds 10000000 cm,
- the list of tiles was empty,

- minimum bathroom width, minimum bathroom length, or tile dimension was zero,
- joint size may be zero.

Sample program runs:

```
Minimum dimensions:
200 300
Tiles:
24 0
18 0
Dimensions: 216.0 x 360.0
```

```
Minimum dimensions:
200 300
Tiles:
7.0 0
16.1 0
10.5 0
Dimensions: 483.0 x 483.0
```

```
Minimum dimensions:
200 50000
Tiles:
7.0 0.5
16.0 0.3
10.1 0.2
Dimensions: 23700.5 x 149618.0
```

```
Minimum dimensions:
200 50000.5
Tiles:
7.0 0.5
16.0 0.3
10.8 0.2
No solution.
```

```
Minimum dimensions:
10 1000000
Tiles:
24.6 0.2
15.8 0.2
16.2 0.4
Dimensions: 28768.2 x 1016800.2
```

```
Minimum dimensions:
10 1000000000
Invalid input.
```

```
Minimum dimensions:
100 100
Tiles:
57.0 abcd
Invalid input.
```

Advice:

- The sample runs above list both the output of your program (bold face text) and user input (regular text). The bold/regular formatting is included here, in the problem statement page, to increase readability of the listing. Your program must output the text without any additional markup.
- Do not forget the newline (\n) after the last output line.
- Data type `double` may be convenient to process the input, however, we recommend a different data type for the core computation.
- The program can be developed without additional functions (i.e. in one big `main`). However, if divided into functions, the program is readable and easier to debug.
- Basic solution of the problem only requires loops and branches. Arrays are not needed.
- The problem offers a bonus tests. The bonus requires a time efficient algorithm. The bonus variant is not recommended for beginners. Even experienced programmers are recommended to start with the basic variant and (optionally) start the optimized version once the basic variant is accepted.
- The bonus test results are huge numbers that do not fit into `int` or `float` data types. A wider data type must be used to successfully pass the bonus test (e.g. a `long long int` data type; do not worry about compiler warnings, the compiler uses `-Wno-long-long` switch). If your implementation does not perform arithmetic operations economically, the limited range of data types may cause problems even in other tests.
- Please strictly adhere to the format of the output. The output must exactly match the output of the reference program. The comparison is done by a machine, the machine requires an exact match. If your program provides output different from the reference, the program is considered malfunctioning. Be very careful, the machine is sensitive event to whitespace characters (spaces, newlines, tabulators). Please note that all output lines are followed by a newline character (\n). This applies even to the last line of the output, moreover, this applies even to the error message. Download the enclosed archive. The archive contains a set of testing inputs and the expected outputs. Read Progtest FAQ to learn how to use input/output redirection and how to simplify testing of your programs.
- Examples of valid input dimensions are `10.5`, `17`, or `24.0`. Input dimensions like `25.06` are invalid (precision greater than 1 mm). Dimensions like `12.00` shall be considered valid. In fact, dimensions like this are not tested, the input processing is a bit easier when dimensions like

`12.00` are accepted. (It would require a custom decimal to binary conversion routine to reject inputs like this.)

- Textual description of valid input data structure is not 100% exact. Therefore we provide a formal specification of the input language in EBNF:

```
    input      ::= { whiteSpace } minSize { whiteSpace } tile
                   { whiteSpace } { tile { whiteSpace } }
    whiteSpace ::= ' ' | '\t' | '\n' | '\r'
    minSize    ::= decimal { whiteSpace } decimal
    tile       ::= decimal { whiteSpace } decimal
    decimal    ::= [ '+' ] integer [ '.' integer [ ( 'e' | 'E' ) [ '+' |
'-' ] integer ] ]  |
                   [ '+' | '-'  ] '.' integer [ ( 'e' | 'E' ) [ '+' | '-' ]
integer ]
    integer    ::= digit { digit }
    digit      ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```