The task is to develop a program to optimize parcel investments.

We assume a grid of land parcels. The size of the grid is known (1 to 2000 rows/columns), moreover, the price of the individual parcels is known. The price is a positive integer. Each parcel is of unit area.

We want to invest money into the parcels. The investment is tricky, however. We may only invest into adjacent parcels that form a rectangular or a square shape. Moreover, we are searching for an investment where the sum of parcel prices less or equal to the invested money. Finally, we want to allocate the largest possible area, given the previous restrictions.

The program must understand two kinds of query. First, a query of type `count investment` finds the total number of possible investments. Second, query of type `list investment` is an extension of `count investment`. It adds the list of the individual parcels that match the query.

The input of the program is:

- the size of the grid (width, height), both dimensions in the range 1 to 2000,
- the price of individual parcels in the grid. The prices are listed in a row-major order in the input,
- the queries.

A query is either `count investment` or `list investment`, where `investment` is the amount of money to invest.

The output of the program are answers to the queries:

- `count investment` answer is the total number of possible parcel allocations such that the allocations form a rectangle/square in the grid, the sum of parcel prices is less than or equal to `investment`, and the total area is the largest possible (given the investment).
- `list investment` results in an answer like the previous query, moreover, it delivers the list of possible parcel allocations. The list of possible allocations consists of lines, each line represents one possible allocation:

  ```
  cost: w x h @ (x,y)
  ```

  where `cost ≤ investment` is the cost of the investment, `w,h` are the dimensions of the allocated parcels, and `x,y` is upper left corner of the allocation.
- If the investment is low, there might not be any parcel available. The answer is different in this case, see sample runs below.

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:

- grid dimensions are not numbers, or are outside the boundaries (1 to 2000),
- parcel price is not an integer, is negative, or zero,

- query is neither 'count' nor 'list',
- investment is not an integer, is negative, is zero, or is missing.

Invest some time to think up the structure of the program. There is an upper limit of the grid size, thus static allocation may be adequate for the task. Next, the searching. A naive searching algorithm may require up to n^6 time, there are improvements that decrease the searching time dramatically. The time limits in the testing environment are very benevolent. A sensible implementation of the naive algorithm passes all but bonus tests.

Sample program runs:

```
Map size:
5 6
Price map:
 13   9  16  14   3
 11   7   5  14   9
  2   5   9   9   4
 11  13   3   8  16
 19  15   9   4   2
  3   7   8  11  27
Queries:
count 20
Max. area: 3 (x 6)
count 50
Max. area: 6 (x 15)
count 190
Max. area: 20 (x 6)
list 1
Not found.
list 2
Max. area: 1 (x 2)
2: 1 x 1 @ (0,2)
2: 1 x 1 @ (4,4)
list 3
Max. area: 1 (x 5)
2: 1 x 1 @ (0,2)
3: 1 x 1 @ (0,5)
3: 1 x 1 @ (2,3)
3: 1 x 1 @ (4,0)
2: 1 x 1 @ (4,4)
list 10
Max. area: 2 (x 3)
7: 2 x 1 @ (0,2)
10: 2 x 1 @ (0,5)
6: 2 x 1 @ (3,4)
list 60
Max. area: 8 (x 1)
60: 4 x 2 @ (0,2)
list 180
Max. area: 20 (x 3)
173: 4 x 5 @ (0,1)
175: 5 x 4 @ (0,1)
174: 4 x 5 @ (1,0)
list 175
Max. area: 20 (x 3)
```

173: 4 x 5 @ (0,1)
175: 5 x 4 @ (0,1)
174: 4 x 5 @ (1,0)
list 173
Max. area: 20 (x 1)
173: 4 x 5 @ (0,1)
list 172
Max. area: 18 (x 3)
165: 3 x 6 @ (0,0)
166: 3 x 6 @ (1,0)
171: 3 x 6 @ (2,0)
count 1000
Max. area: 30 (x 1)

Map size:
12 1
Price map:
1 2 3
4 1 2 3 4 1 2
3 4
Queries:
count 10
Max. area: 4 (x 9)
list 9
Max. area: 3 (x 10)
6: 3 x 1 @ (0,0)
9: 3 x 1 @ (1,0)
8: 3 x 1 @ (2,0)
7: 3 x 1 @ (3,0)
6: 3 x 1 @ (4,0)
9: 3 x 1 @ (5,0)
8: 3 x 1 @ (6,0)
7: 3 x 1 @ (7,0)
6: 3 x 1 @ (8,0)
9: 3 x 1 @ (9,0)
count 8
Max. area: 3 (x 7)
list 12
Max. area: 5 (x 4)
11: 5 x 1 @ (0,0)
12: 5 x 1 @ (1,0)
11: 5 x 1 @ (4,0)
12: 5 x 1 @ (5,0)
list 0
Invalid input.

Map size:
2 2
Price map:
1 2 3 test
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.
- Use `int` data type to represent the prices.
- Coordinates (0,0) correspond to the upper left corner, `x` coordinate grows to the right, `y` coordinate grows downwards.
- There is no need to dynamically allocate memory in this homework. Memory limits are big enough to allocate the price maps statically. However, there may not be enough space in the stack to allocate the price maps as local variables. An allocation in the data segment may solve the problem, see `static` keyword.
- The newlines in the input price maps do not have to follow the shape of the grid. In fact, the program may ignore the input newlines since the dimensions of the grid are known in advance.
- The number of parcel allocations may be huge in the bonus tests. The printing of all matching parcel allocations slows down the program. Therefore, the bonus tests use mostly `count` queries for huge inputs.
- There is not any exact rule how to order of the lines in the `list` answers. Your implementation may print the allocations in any order, the testing environment fixes the order before the comparison. For example input:

```
Map size:
6 4
Price map:
1 2 1 2 1 2
2 1 2 1 2 1
1 2 1 2 1 2
2 1 2 1 2 1
Queries:
list 3
```

  may result in:

```
Max. area: 2 (x 38)
3: 2 x 1 @ (0,0)
3: 2 x 1 @ (1,0)
3: 2 x 1 @ (2,0)
3: 2 x 1 @ (3,0)
3: 2 x 1 @ (4,0)
3: 1 x 2 @ (0,0)
3: 1 x 2 @ (1,0)
3: 1 x 2 @ (2,0)
3: 1 x 2 @ (3,0)
3: 1 x 2 @ (4,0)
3: 1 x 2 @ (5,0)
3: 2 x 1 @ (0,1)
3: 2 x 1 @ (1,1)
3: 2 x 1 @ (2,1)
```

```
3: 2 x 1 @ (3,1)
3: 2 x 1 @ (4,1)
3: 1 x 2 @ (0,1)
3: 1 x 2 @ (1,1)
3: 1 x 2 @ (2,1)
3: 1 x 2 @ (3,1)
3: 1 x 2 @ (4,1)
3: 1 x 2 @ (5,1)
3: 2 x 1 @ (0,2)
3: 2 x 1 @ (1,2)
3: 2 x 1 @ (2,2)
3: 2 x 1 @ (3,2)
3: 2 x 1 @ (4,2)
3: 1 x 2 @ (0,2)
3: 1 x 2 @ (1,2)
3: 1 x 2 @ (2,2)
3: 1 x 2 @ (3,2)
3: 1 x 2 @ (4,2)
3: 1 x 2 @ (5,2)
3: 2 x 1 @ (0,3)
3: 2 x 1 @ (1,3)
3: 2 x 1 @ (2,3)
3: 2 x 1 @ (3,3)
3: 2 x 1 @ (4,3)
```

or:

```
Max. area: 2 (x 38)
3: 1 x 2 @ (0,0)
3: 1 x 2 @ (1,0)
3: 1 x 2 @ (2,0)
3: 1 x 2 @ (3,0)
3: 1 x 2 @ (4,0)
3: 1 x 2 @ (5,0)
3: 2 x 1 @ (0,0)
3: 2 x 1 @ (1,0)
3: 2 x 1 @ (2,0)
3: 2 x 1 @ (3,0)
3: 2 x 1 @ (4,0)
3: 1 x 2 @ (0,1)
3: 1 x 2 @ (1,1)
3: 1 x 2 @ (2,1)
3: 1 x 2 @ (3,1)
3: 1 x 2 @ (4,1)
3: 1 x 2 @ (5,1)
3: 2 x 1 @ (0,1)
3: 2 x 1 @ (1,1)
3: 2 x 1 @ (2,1)
3: 2 x 1 @ (3,1)
3: 2 x 1 @ (4,1)
3: 1 x 2 @ (0,2)
3: 1 x 2 @ (1,2)
3: 1 x 2 @ (2,2)
3: 1 x 2 @ (3,2)
3: 1 x 2 @ (4,2)
3: 1 x 2 @ (5,2)
3: 2 x 1 @ (0,2)
3: 2 x 1 @ (1,2)
```

```
3: 2 x 1 @ (2,2)
3: 2 x 1 @ (3,2)
3: 2 x 1 @ (4,2)
3: 2 x 1 @ (0,3)
3: 2 x 1 @ (1,3)
3: 2 x 1 @ (2,3)
3: 2 x 1 @ (3,3)
3: 2 x 1 @ (4,3)
```

or any other of the remaining 523022617466601111760007224100074291199999998 permutations.

- 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 } gridSize { whiteSpace } priceMap
{ whiteSpace } queryList
    whiteSpace ::= ' ' | '\t' | '\n' | '\r'
    gridSize   ::= integer { whiteSpace } integer
    priceMap   ::= integer { { whiteSpace } integer }
    queryList  ::= { query { whiteSpace } }
    query      ::= ( 'list' | 'count' ) { whiteSpace } integer
    integer    ::= [ '+' ] digit { digit }
    digit      ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```