Problem A: Longest Continuous Ordered Subsequence with Maximal Summation

Description

A numeric sequence of a_i is ordered if $a_1 <= a_2 <= ... <= a_N$. Let the continuous subsequence of the given numeric sequence (a_1, a_2, ..., a_N) be any sequence (a_{i1}, a_{i2}, ..., a_{iK}), where i_1, i_2, ..., i_K are continuous numbers. In this problem, your job is to find the length of the longest continuous ordered subsequence and calculate the summation of the numbers. If more than one longest continuous ordered subsequence has the same length, you must choose the one with the maximal summation.

For example, sequence (1, 3, 3, 5, 4, 9, 9) has continuous ordered subsequences, e. g., (1, 3, 3, 5), (4, 9, 9) and many others. The longest continuous ordered subsequence is of length 4, (1, 3, 3, 5), with the summation of 12.

Specification

You must implement the FindLCOS class with the following public member functions:

Member functions	Description
FindLCOS(int)	Constructor. The int parameter means the total number of elements in the sequence.
~FindLCOS()	Destructor.
void push(int)	Push elements into class.
void show()	Print the elements of the longest continuous ordered subsequence with Maximal Summation. Each element separated by a space.
int length()	Return the length of the longest continuous ordered subsequence with Maximal Summation.
int sum()	Return the summation of the longest continuous ordered subsequence with Maximal Summation.

Sample Main function

```
int main()
{
	int n;
	cin >> n;
	FindLCOS flcos(n);
	for (int i=0;i<n;i++) {
		int element;
		cin >> element;
		flcos.push(element);
	}
	flcos.show();
	cout << endl << flcos.length() << endl;
	cout << flcos.sum() << endl;
	return 0;
}
```

Input

The first line of input contains an integer N indicating the length of the numeric sequence in this test case. The second line contains the elements of sequence a1 a2 ... aN , where ai is in the range of integer value, and each number is separated by a space.

Output

The first line should output the longest continuous subsequence with maximal summation. The second line should output the length of the subsequence. The third line should output the sum of the subsequence.

Sample input	**Sample output**
7 1 2 3 1 2 4 3	1 2 4 3 7

Restriction

1. The code you submitted should only contain the FindLCOS class (with no header file and main function.)
2. In the question, the <iostream> is the only header file allowed.

Problem B: Big String

Description

You have to implement a class called BigString. The class should be able to achieve some functions for string such as extend, append, replace…etc. For example, giving "AAA" + "BBB", the new string is "AAABBB". The functions of BigString should be implemented exactly as required.

Note that STL is prohibited in this problem. However, you may use the cstring library if needed.

Specification

The following functions should be implemented, and only these functions should be visible from outside the class:

Functions	Description
void set(char*)	Set the current string.
char* show()	Return the current string.
int length()	Return the current string length.
BigString operator+ (const BigString &)	Append the current string of (BigString) to current string
BigString operator* (int)	Extend the current string by itself (int)-times.
BigString operator- (const BigString &)	Delete substrings which are as same as the current string of (BigString)
void operator= (const BigString &)	Assignment operator
void replace(const BigString , const BigString)	Replace substrings which are as same as the current string of th first (BigString) by the current string of the second (BigString)

Example

```
int main(){
    BigString a,b,c,d,e;
    a.set("OOP");
    b.set("PROBLEM");
    c.set("OP");
    d.set("O");
    e=a+b;
    cout<<e.show()<<endl;
```

```
        e=e-c;
        cout<<e.show()<<endl;
        c.set("Q_Q");
        e.replace(d,c);
        cout<<e.show()<<endl;
        cout<<e.length()<<endl;
        e=e*2;
        cout<<e.show()<<endl;
        cout<<e.length()<<endl;
        return 0;
}
```

Result

```
OOPPROBLEM
OPROBLEM
Q_QPRQ_QBLEM
12
Q_QPRQ_QBLEMQ_QPRQ_QBLEM
24
```

Restriction

1. The code you submitted should only contain the BigString class (with no header file and main function.)
2. In the question, the <iostream> and <cstring> are the only header files allowed.

Problem C: Chamber Escape - risky slide

Description

You're locked in a secret chamber, it's a simple rectangle room with a tiled floor and no furniture or contents of any kind, You will practice the risky slide by starting out standing at any tile and running to other tile parallel to one of the walls of the chamber, and at the transition between some pair of tiles you have two choices: running or start sliding. The escape requirement is to achieve the longest possible slide, with distance measured as the number of tiles you completely traverse during the slide (partial traversal doesn't count).

This would be an easy problem, but the style of the tiles in chamber introduces a complication. Each tile has a particular stickiness that affects both how well you can run and how well you can slide on it. Each tile has a stickiness rating between 1 and 9, inclusive. The stickiness affects your movement as follows:

- Running across a tile with stickiness K grants K units of energy.
- Sliding across a tile with stickiness K reduce you K units of energy.

For example, a tile with stickiness 9 is not slippery at all (it reduce 9 energy) and you can move across it very efficiently (it grants 9 energy)

The energy start from the tile you stand on, if a tile would reduce your energy below 0, you stop sliding somewhere in the middle of the tile and fail to traverse it completely. In reality, run-up and slide must be a single line.

Input

Your input will consist of an arbitrary number of test cases. Each case begins with a line containing integers R and C, the number of rows and columns respectively, of tiles in the chamber. This will be followed by R lines, each containing a string of length C, describing the layout of the tiles in chamber. The value of each element in these strings is the stickiness as defined above of the corresponding tile.

Output

Output, for each test case and separated by newlines, the maximal possible number of complete tiles that you can traverse in a single slide.

Constraints

$1 \le R, C \le 50$

Run? Slide?

Example input

```
1 1
5
2 3
112
324
5 5
55555
51115
51915
51115
55555
1 4
5222
1 5
52222
```

Example output

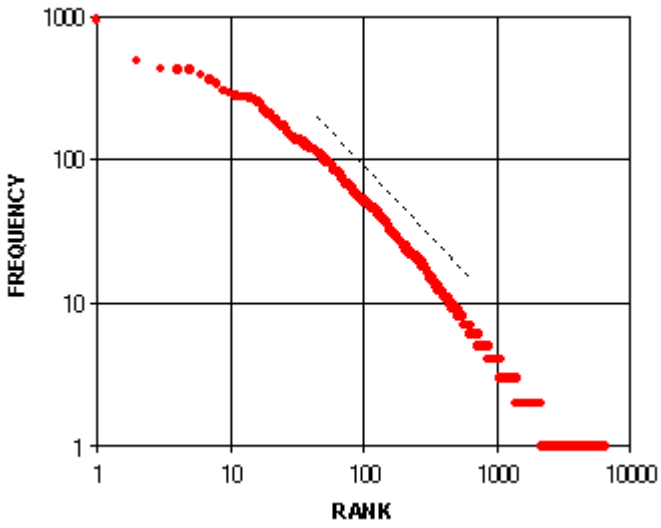
```
0
2
3
2
3
```

Problem D: Zipf's Law

Description

Harvard linguistics professor George Kingsley Zipf (1902-1950) observed that the frequency of the *k*th most common word in a text is roughly proportional to 1/*k*. He justified his observations in a book titled *Human behavior and the principle of least effort* published in 1949. While Zipf's rationale has largely been discredited, the principle still holds, and others have afforded it a more sound mathematical basis.

You are to find all the words occurring *n* times in an English text. A word is a sequence of letters. Words are separated by non-letters. Capitalization should be ignored. A word can be of any length that an English word can be.



Input

Input consists of several test cases. The first line of each case contains a single positive integer *n*. Several lines of text follow which will contain no more than 10000 words. The text for each case is terminated by a single line containing EndOfText. EndOfText does not appear elsewhere in the input and is not considered a word.

Output

For each test case, output the words which occur *n* times in the input text, one word per line, lower case, in alphabetical order. If there are no such words in input, output the following line:
There is no such word.
Leave a blank line between cases.

Sample Input

2

In practice, the difference between theory and practice is always
greater than the difference between theory and practice in theory.
- Anonymous

Man will occasionally stumble over the truth, but most of the
time he will pick himself up and continue on.
- W. S. L. Churchill
EndOfText

Output for Sample Input

between
difference
in
will

Problem E: The PageRank Algorithm

Description

PageRank is a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page. PageRank can be calculated for collections of documents of any size. It is assumed in several research papers that the distribution is evenly divided among all documents in the collection at the beginning of the computational process. The PageRank computations require several passes, called "iterations", through the collection to adjust approximate PageRank values to more closely reflect the theoretical true value.

A probability is expressed as a numeric value between 0 and 1. A 0.5 probability is commonly expressed as a "50% chance" of something happening. Hence, a PageRank of 0.5 means there is a 50% chance that a person clicking on a random link will be directed to the document with the 0.5 PageRank.

Assume a small universe of four web pages: **A**, **B**, **C** and **D**. The initial approximation of PageRank would be evenly divided between these four documents. Hence, each document would begin with an estimated PageRank of 0.25.

In the original form of PageRank initial values were simply 1. This meant that the sum of all pages was the total number of pages on the web. Later versions of PageRank (see the formulas below) would assume a probability distribution between 0 and 1. Here a simple probability distribution will be used—hence the initial value of 0.25.

If pages **B**, **C**, and **D** each only link to **A**, they would each confer 0.25 PageRank to **A**. All PageRank **PR( )** in this simplistic system would thus gather to **A** because all links would be pointing to **A**.

$$PR(A) = PR(B) + PR(C) + PR(D).$$

This is 0.75.

Suppose that page **B** has a link to page **C** as well as to page **A**, while page **D** has links to all three pages. The *value of the link-votes is divided among all the outbound links on a page*. Thus, page **B** gives a vote worth 0.125 to page **A** and a vote worth 0.125 to page **C**. Only one third of **D**'s PageRank is counted for A's PageRank (approximately 0.083).

$$PR(A) = \frac{PR(B)}{2} + \frac{PR(C)}{1} + \frac{PR(D)}{3}.$$

In other words, the PageRank conferred by an outbound link is equal to the document's own PageRank score divided by the normalized number of outbound links **L( )** (it is assumed that links to specific URLs only count once per document).

$$PR(A) = \frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)}.$$

In the general case, the PageRank value for any page **u** can be expressed as:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)},$$

i.e. the PageRank value for a page **u** is dependent on the PageRank values for each page **v** out of the set $\mathbf{B_u}$ (this set contains all pages linking to page **u**), divided by the number *L*(*v*) of links from page **v**.

The PageRank theory holds that even an imaginary surfer who is randomly clicking on links will eventually stop clicking. The probability, at any step, that the person will continue is a damping factor *d*. Various studies have tested different damping factors, but it is generally assumed that the damping factor will be set around 0.85.

The damping factor is subtracted from 1 (and in some variations of the algorithm, the result is divided by the number of documents (*N*) in the collection) and this term is then added to the product of the damping factor and the sum of the incoming PageRank scores. That is,

$$PR(A) = \frac{1-d}{N} + d\left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \cdots\right).$$

When calculating PageRank, pages with no outbound links are assumed to link out to all other pages in the collection. Their PageRank scores are therefore divided evenly among all other pages. In other words, to be fair with pages that are not sinks, these random transitions are added to all nodes in the Web, with a residual probability of usually *d* = 0.85, estimated from the frequency that an average surfer uses his or her browser's bookmark feature.

So, the equation is as follows:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

where $p_1,p_2,...,p_N$ are the pages under consideration, $M(p_i)$ is the set of pages that link to p_i, $L(p_j)$ is the number of outbound links on page p_j, and N is the total number of pages.

The PageRank values are the entries of the dominant eigenvector of the modified adjacency matrix. This makes PageRank a particularly elegant metric: the eigenvector is

$$\mathbf{R} = \begin{bmatrix} PR(p_1) \\ PR(p_2) \\ \vdots \\ PR(p_N) \end{bmatrix}$$

where **R** is the solution of the equation

$$\mathbf{R} = \begin{bmatrix} (1-d)/N \\ (1-d)/N \\ \vdots \\ (1-d)/N \end{bmatrix} + d \begin{bmatrix} \ell(p_1,p_1) & \ell(p_1,p_2) & \cdots & \ell(p_1,p_N) \\ \ell(p_2,p_1) & \ddots & & \vdots \\ \vdots & & \ell(p_i,p_j) & \\ \ell(p_N,p_1) & \cdots & & \ell(p_N,p_N) \end{bmatrix} \mathbf{R}$$

where the adjacency function $\ell(p_i,p_j)$ is 0 if page p_j does not link to p_i, and normalized such that, for each j

$$\sum_{i=1}^{N} \ell(p_i,p_j) = 1$$

,

i.e. the elements of each column sum up to 1, so the matrix is a stochastic matrix (for more details see the computation section below). Thus this is a variant of the eigenvector centrality measure used commonly in network analysis.

Because of the large eigengap of the modified adjacency matrix above, the values of the PageRank eigenvector are fast to approximate (only a few iterations are needed).

As a result of Markov theory, it can be shown that the PageRank of a page is the probability of being at that page after lots of clicks. This happens to equal t^{-1} where t is the expectation of the number of clicks (or random jumps) required to get from the page back to itself.

In this problem, you will implement an easy version of the PageRank algorithm. You will be given with all the information needed, you just need to calculate the result.

Input

The first line will be N, then followed by N lines of inputs, standing for the N webpages in order. For each line, there will also be N numbers of input, either 0 or 1, to show the "vote" to other pages. To ease the problem, N will not be over 100.

Output

You should output the PageRank for each webpage, with an endline after each line. The number should accurate to only the second digit after the decimal point.

Sample Input

4

0 1 1 1

0 0 0 1

1 1 0 0

1 0 0 0

Sample Output

0.36

0.21

0.11

0.32