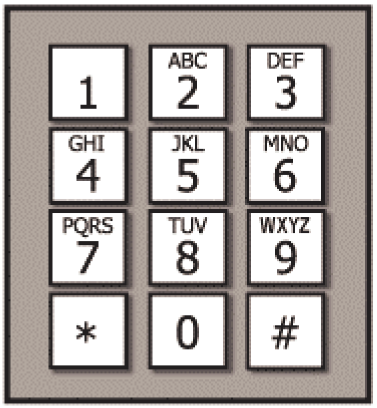
Problem S1: Snow Calls

You've been snowed in at your summer residence. And without the Internet! Unfortunately, this means you're going to have rely on using the phone to get what you need to survive: pizza, pop, and the latest video games.

Often times, companies replace the digits in their phone numbers with characters to make their phone numbers more memorable. Because apparently, it's easier to remember 416-BUY-MORE than it is to remember 416-289-6673. Some companies even add extra digits or characters (like 604-PIZZABOX) but any digits after the 10th are irrelevant.

Since it's getting tedious to do the conversion by hand, write a program to help change all the phone numbers in your phone book to the form *xxx-xxx-xxxx*, using the below image to assist you.



Input
Input consists of a series of test cases. The first line consists of an integer *t*, the number of test cases. Following this are *t* lines consisting of alpha-numeric characters separated by hyphens, representing valid phone numbers. No line is longer than 40 characters. Input will be contained in the file `s1.in`.

Output
For each test case, output the phone number in the form *xxx-xxx-xxxx* to the screen.

Sample Input
```
5
88-SNOW-5555
519-888-4567
BUY-MORE-POP
416-PIZZA-BOX
5059381123
```

Sample Output
```
887-669-5555
519-888-4567
289-667-3767
416-749-9226
505-938-1123
```

Problem S2: Mouse Move

Most likely, you will notice that you have a *mouse* attached to your computer, which lets you move the *cursor* around the *screen*. Your job is to get between the mouse and the cursor.

Suppose that the bottom left-hand corner of your screen is (0,0), and all points on the screen are given by integer co-ordinates (*x*, *y*) where *0* <= *x* <= *c* and *0* <= *y* <= *r*. Thus, the top-right corner of the screen is at position (*c*, *r*), bottom-right corner is (*c*, 0), and top-left corner is (0, *r*).

When a mouse is moved, it sends a pair of integers (*a*, *b*), indicating that the cursor should be moved *a* units in the *x*-direction and *b* units in the *y*-direction. It is worth noting that this is *relative* motion (i.e., how far to move) rather than *absolute* motion (i.e., where to move). It is also worth noting that *a* and *b* may be positive, negative or zero.

You can assume the mouse starts at position (0,0). You job is to read input messages (i.e., relative motion positions sent by the mouse) and update the cursor to the new position on the screen. Your output (to the screen) will be the position of the mouse after each move.

If the mouse hits the screen boundary, it stops moving in *that direction*. For example, if the mouse is supposed to move (-100, -10) from its current position (30, 40), the final positions will be (0, 30): the mouse will hit the left-hand side boundary, but still manages to move down.

Input is listed as pairs, the first pair being (*c*, *r*), followed by the relative motion pairs (*x*, *y*). The input is terminated when the mouse moves (0,0). The input will be contained in the file `s2.in`.

Sample Input 1
```
100 200
10 40
-5 15
30 –30
0 0
```

Sample Output for Sample Input 1
```
10 40
5 55
35 25
```

Sample Input 2
```
30 40
30 40
-100 -10
0 0
```

Sample Output for Sample Input 2
```
30 40
0 30
```

Problem S3: Quantum Operations

Quantum computing is currently a hot topic in research. If they can be built, quantum computers will have the ability to perform certain computing tasks much faster than any computer in existence today. Fortunately, you won't need a quantum computer to do this question.

A fundamental concept in quantum computing is the idea of as a quantum operation. A quantum operation can be essentially thought of as a matrix. Also, if you perform two quantum operations in parallel on separate quantum data, this can be thought of a larger quantum operation. Thinking of these operations in terms of matrices again, the resulting matrix from performing two matrices in parallel is called the *tensor product* of the two matrices (using the symbol $\otimes$). This is different than the normal product of matrices that you may have learned about.

In a tensor product, you are given two matrices (which are essentially two-dimensional arrays). We will call them *A* and *B*, and we will represent the individual elements of these two matrices this way:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1q} \\ b_{21} & b_{22} & \cdots & b_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ b_{p1} & b_{p2} & \cdots & a_{pq} \end{bmatrix}.$$

Notice that the size of matrix *A* is $m \times n$ (*m* rows and *n* columns), and the size of *B* is $p \times q$.

The tensor product of these two matrices will be an $mp \times nq$ matrix (with *mp* rows and *nq* columns) that looks like:

$$A \otimes B = \begin{bmatrix} a_{11}[B] & a_{12}[B] & \cdots & a_{1n}[B] \\ a_{21}[B] & a_{22}[B] & \cdots & a_{2n}[B] \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}[B] & a_{m2}[B] & \cdots & a_{mn}[B] \end{bmatrix},$$

where $a_{ij}[B]$ simply indicates that each element in *B* is being multiplied by a_{ij}.
Finally notice that the tensor product is not *commutative*, which means that changing the order of matrices may change the answer ($A \otimes B \neq B \otimes A$).

For more than two matrices, we will define $A \otimes B \otimes C = (A \otimes B) \otimes C$, although it does not technically matter, since the tensor product is *associative*.

Your task is to compute and output the tensor product of two or more given matrices.

Input

The first line of input contains the number of matrices, *N*, a positive integer. Then, there are *N* blocks of lines describing the matrices in order.

In each block, the first line will contain two positive integers, *r* and *c*, separated by a space, indicating the number of rows and columns, respectively. Then, the next *r* lines will denote the rows, in order. Each line will contain *c* integers, separated by spaces. Input is contained in the file `s3.in`.

Output

The output (to the screen) will be 6 integers in the following order:

- the maximum element in the tensor product
- the minimum element in the tensor product
- the maximum row sum (i.e., sum of entries in each row)
- the minimum row sum
- the maximum column sum
- the minimum column sum

You may assume that tensor product matrix will have no more than 1024 rows and no more than 1024 columns.

Input	**Output**	**Actual Tensor Product**

`2`
`2 2`
`1 1`
`1 -1`
`2 2`
`1 0`
`0 1`
	`1`
	`-1`
	`2`
	`0`
	`2`
	`0`
		`1 0 1 0`
		`0 1 0 1`
		`1 0 -1 0`
		`0 1 0 -1`

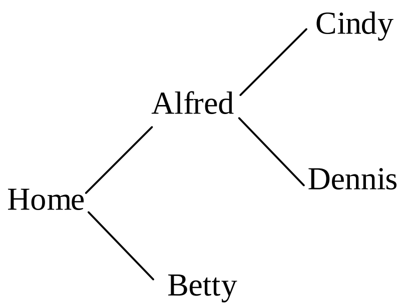
`3`
`2 2`
`1 0`
`0 3`
`2 2`
`1 1`
`1 -1`
`2 2`
`1 0`
`0 1`
	`3`
	`-3`
	`6`
	`0`
	`6`
	`0`
		`1 0 1 0 0 0 0 0`
		`0 1 0 1 0 0 0 0`
		`1 0 -1 0 0 0 0 0`
		`0 1 0 -1 0 0 0 0`
		`0 0 0 0 3 0 3 0`
		`0 0 0 0 0 3 0 3`
		`0 0 0 0 3 0 -3 0`
		`0 0 0 0 0 3 0 -3`

Problem S4: Pyramid Message Scheme

Spamway Inc. maintains a network of zombie computers to solicit and collect orders for its various fine products. Each zombie computer is responsible for zero or more subordinate zombies that it coordinates in these activities.

Spamway uses a simple communication strategy among its zombies for transmitting solicitations and receiving orders. Each solicitation originates at Spamway's head zombie, which then communicates it to each of its subordinates in turn, waiting to collect orders from one subordinate before proceeding to the next. Each subordinate employs the same strategy - it sends to and receives from each of its subordinates in turn.

For example, suppose that Home has two subordinate zombies named Alfred and Betty; Alfred's subordinates are named Cindy and Dennis; Betty has no subordinates. This organization is pictured below.



Home first sends to Alfred; Alfred then sends to Cindy; Cindy responds to Alfred; Alfred sends to Dennis; Dennis responds to Alfred; Alfred responds to Home; Home sends to Betty; Betty responds to Home.

Each message takes 10 seconds to be delivered. So the example given above would be completed in 80 seconds. You have been retained by Spamway, who will pay you handsomely (in Spam Bucks which may be redeemed for any of their valuable products) to help them reduce the time necessary to solicit and collect orders. In particular, Spamway is considering a new strategy in which each zombie sends out messages to each of its subordinates and waits for their responses only after all messages have been sent.

Spamway's network administrator has captured a chronological list of the name of the recipient of each message involved in a particular solicitation. For the example above, using the slow strategy, this list would be: Alfred, Cindy, Alfred, Dennis, Alfred, Home, Betty, Home. (Note that 8 messages at 10 seconds per message is 80 seconds.)

Using the new and improved strategy, Home sends to Alfred and Betty simultaneously, Alfred sends to Cindy and Dennis at the same time as Betty is responding, Cindy and Dennis respond simultaneously to Alfred and finally Alfred responds to Home. Using the new strategy, Spamway needs only 40 seconds to accomplish the communication that takes 80 seconds using the old strategy. Thus, Spamway can send twice as many solicitations and make twice as much money.

Input

As input, you are given lists of names describing the order that messages are received using the old Spamway strategy. The input contains the integer *L*, followed by *L* message lists. Each list begins with an integer, *n*, identifying the number of message recipients in the list, followed by *n* lines, each containing the name of a message recipient. Input is contained in the file `s4.in`.

Output

For each list you are to print out a single integer indicating the amount of time in seconds that Spamway saves.

Sample Input

```
1
8
Alfred
Cindy
Alfred
Dennis
Alfred
Home
Betty
Home
```

Output for Sample Input

```
40
```

Problem S5: Pinball Ranking

Pinball is an arcade game in which an individual player controls a silver ball by means of flippers, with the objective of accumulating as many *points* as possible. At the end of each game, the player's score and rank are displayed. The score, an integer between 0 and 1 000 000 000, is that achieved by the player in the game just ended. The rank is displayed as "*r* of *n*". *n* is the total number of games ever played on the machine, and *r* is the position of the score for the just-ended game within this set. More precisely, *r* is one greater than the number of games whose score exceeds that of the game just ended.

You are to implement the pinball machine's ranking algorithm. The first line of input contains a positive integer, *t*, the total number of games played in the lifetime of the machine. *t* lines follow, given the scores of these games, in chronological order. Input is contained in the file `s5.`in.

You are to output the average of the ranks (rounded to two digits after the decimal) that would be displayed on the board.

At least one test case will have *t* <= 100. All test cases will have *t* <= 100 000.

Sample Input

```
5
100
200
150
170
50
```

Output for Sample Input

```
2.20
```

Explanation for Sample Output

The pinball screen would display (in turn):

```
1 of 1
1 of 2
2 of 3
2 of 4
5 of 5
```

The average rank is (1+1+2+2+5)/5 = 2.20.