Exercises: Multidimensional Arrays

You can check your solutions here: https://judge.softuni.bg/Contests/3173/Multidimentional-Arrays.

1. 2X2 Squares in Matrix

Find the count of **2 x 2 squares of equal chars** in a matrix of characters.

Input

- On the **first line**, you are given the integers **rows** and **cols –** the matrix's dimensions
- Matrix characters come at the next **rows** lines (space separated)

Output

- Print **the count** of all the squares matrices you have found.

Examples

Input	Output	Comments
3 4 A B B D E B B B I J B B	2	Two 2 x 2 squares of equal cells: A **B B** D A B B D E **B B** B E B **B B** I J B B I J **B B**
2 2 a b c d	0	No 2 x 2 squares of equal cells exist.

2. Maximal Sum 3X3

Write a program that reads a rectangular integer matrix of size **N x M** and finds in it the **square 3 x 3** that **has maximal sum of its elements**.

Input

- On the first line, you will receive the rows **N** and columns **M**.
- On the next **N lines** you will receive **each row with its columns**

Output

- Print the **elements** of the 3 x 3 square as a matrix, along with their **sum**

Examples

Input	Matrix	Output
4 5 1 5 5 2 4 2 1 4 14 3 3 7 11 2 8 4 8 12 16 4	1 5 5 2 4 2 **1 4 14** 3 3 **7 11 2** 8 4 **8 12 16** 4	Sum = 75 1 4 14 7 11 2 8 12 16

3. Snake Moves

You are walking in the park and you encounter a snake! You are terrified, and you start running zig-zag, so the snake starts following you.

You have a task to visualize the snake's path in a square form. A **snake** is represented by **a string**. The **isle** is a **rectangular matrix of size NxM**. A snake starts going down from the **top-left corner** and slithers its way down. The first cell is filled with the first symbol of the snake, the second cell is filled with the second symbol, etc. The snake is as long as it takes in order to **fill the stairs completely** – if you reach the end of the string representing the snake, start again at the beginning. After you fill the matrix with the snake's path, you should print it.

Input

- The input data should be read from the console. It consists of exactly two lines
- On the first line, you'll receive the **dimensions** of the stairs in format: **"N M"**, where **N** is the number of **rows**, and **M** is the number of **columns**. They'll be separated by a single space
- On the second line you'll receive the string representing the **snake**

Output

- The output should be printed on the console. It should consist of **N lines**
- Each line should contain a string representing the respective row of the matrix

Constraints

- The **dimensions** N and M of the matrix will be integers in the range [1 … 12]
- The **snake** will be a string with length in the range [1 … 20] and **will not contain any whitespace characters**

Examples

Input	Output
5 6 SoftUni	SoftUn UtfoSi niSoft foSinU tUniSo

4. Jagged Array Manipulator

Create a program that populates, analyzes and manipulates the elements of a matrix with unequal length of its rows.

First you will receive an **integer N** equal to the **number of rows** in your matrix.

On the **next N lines**, you will receive **sequences of integers**, **separated** by a single **space**. Each sequence is a **row** in the matrix.

After populating the matrix, start analyzing it. If a **row** and the **one below** it have **equal length**, **multiply** each **element** in **both** of them by **2**, **otherwise** - **divide** by **2**.

Then, you will receive commands. There are three possible commands:

- "**`Add {row} {column} {value}`**" - **add `{value}`** to the element at the **given indexes**, if they are **valid**
- "**`Subtract {row} {column} {value}`**" - **subtract `{value}`** from the element at the **given indexes**, if they are **valid**
- "**End**" - print the **final state** of the **matrix** (all elements **separated by a single space**) and **stop** the program

Input

- On the first line, you will receive the **number of rows** of the matrix - integer **N**
- On the next **N** lines, you will receive **each row** - **sequence of integers**, separated by a single **space**
- **`{value}`** will always be **integer** number
- Then you will be receiving commands until reading "**End**"

Output

- The output should be printed on the console and it should consist of **N lines**
- Each line should contain a string representing the **respective row** of the **final matrix**, elements **separated** by a single **space**

Constraints

- The **number of rows** N of the matrix will be integer in the range [2 … 12]
- The **input** will always **follow** the **format above**
- **Think about data types**

Examples

Input	Output
5 10 20 30 1 2 3 2 2 10 10 End	20 40 60 1 2 3 2 2 5 5
5 10 20 30 1 2 3 2 2 10 10 Add 0 10 10 Add 0 0 10 Subtract -3 0 10 Subtract 3 0 10 End	30 40 60 1 2 3 2 -8 5 5

5. Knight Game

Chess is the oldest game, but it is still popular these days. For this task we will use only one chess piece – the **Knight**.

The knight moves to the **nearest** square but **not on the same row**, **column**, or **diagonal**. (This can be thought of as moving two squares horizontally, then one square vertically, or moving one square horizontally then two squares vertically— i.e. in an "**L**" pattern.)

The knight game is played on a board with dimensions **N x N** and a lot of chess knights **0 <= K <= N^2**.

You will receive a board with **K** for knights and '**0**' for empty cells. Your task is to remove a minimum of the knights, so there will be no knights left that can attack another knight.

Input

- On the first line, you will receive the **N** size of the board
- On the next **N** lines, you will receive strings with **Ks** and **0s**.

Output

Print a single integer with the minimum number of knights that needs to be removed

Constraints

- Size of the board will be 0 < N < 30
- Time limit: 0.3 sec. Memory limit: 16 MB.

Examples

Input	Output
5 0K0K0 K000K 00K00 K000K 0K0K0	1
2 KK KK	0
8 0K0KKK00 0K00KKKK 00K0000K KKKKKK0K K0K0000K KK00000K 00K0K000 000K00KK	12

6. *Bombs

You will be given a square matrix of integers, each integer separated by a **single space**, and each row on a new line. Then on the last line of input you will receive indexes - coordinates to several cells separated by a **single space**, in the following format: **row1,column1 row2,column2 row3,column3…**

On those cells there are bombs. You have to proceed **every bomb**, one by one in the order they were given. When a bomb explodes deals damage **equal** to its **own integer value**, to **all** the cells **around** it (in every direction and in all diagonals). One bomb can't explode more than once and after it does, its value becomes **0**. When a cell's value reaches **0 or below**, **it dies**. Dead cells **can't explode**.

You must **print the count of all alive cells** and **their sum**. Afterwards, print the matrix with all of its cells (including the dead ones).

Input

- On the first line, you are given the integer N – the size of the square matrix.
- The next N lines holds the values for every row – N numbers separated by a space.
- On the last line you will receive the coordinates of the cells with the bombs in the format described above.

Output

- On the first line you need to print the count of all alive cells in the format:
  "**`Alive cells: {aliveCells}`**"
- On the second line you need to print the sum of all alive cell in the format:

"`Sum: {sumOfCells}`"

- In the end print the matrix. The cells must be **separated by a single space**.

Constraints

- The size of the matrix will be between **[0…1000].**
- The bomb coordinates will **always** be in the matrix.
- The bomb's values will always be **greater** than **0**.
- The integers of the matrix will be in range **[1…10000].**

Examples

Input	Output	Comments
4 8 3 2 5 6 4 7 9 9 9 3 6 6 8 1 2 1,2 2,1 2,0	Alive cells: 3 Sum: 12 8 -4 -5 -2 -3 -3 0 2 0 0 -4 -1 -3 -1 -1 2	First the bomb with value 7 will explode and reduce the values of the cells around it. Next the bomb with coordinates 2,1 and value 2 (initially 9-7) will explode and reduce its neighbour cells. In the end the bomb with coordinates 2,0 and value 7 (initially 9-2) will explode. After that you have to print the count of the alive cells, which is 3, and their sum is 12. Print the matrix after the explosions.
3 7 8 4 3 1 5 6 4 9 0,2 1,0 2,2	Alive cells: 3 Sum: 8 4 1 0 0 -3 -8 3 -8 0	

7. *Radioactive Mutant Vampire Bunnies

Browsing through GitHub, you come across an old JS Basics teamwork game. It is about very nasty bunnies that multiply extremely fast. There's also a player that has to escape from their lair. You really like the game, so you decide to port it to C# because that's your language of choice. The last thing that is left is the algorithm that decides if the player will escape the lair or not.

First, you will receive a line holding integers **N** and **M**, which represent the rows and columns in the lair. Then you receive **N** strings that can **only** consist of **"."**, **"B"**, **"P"**. The **bunnies** are marked with "**B",** the **player** is marked with "**P**", and **everything** else is free space, marked with a dot **"."**. They represent the initial state of the lair. There will be **only** one player. Then you will receive a string with **commands** such as **LLRRUUDD** – where each letter represents the next **move** of the player (Left, Right, Up, Down).

After each step of the player, each of the bunnies spread to the up, down, left and right (neighboring cells marked as "." **changes** their value to B). If the player **moves** to a bunny cell or a bunny **reaches** the player, the player has died. If the player goes **out** of the lair **without** encountering a bunny, the player has won.

When the player **dies** or **wins**, the game ends. All the activities for **this** turn continue (e.g. all the bunnies spread normally), but there are no more turns. There will be **no** stalemates where the moves of the player end before he dies or escapes.

Finally, print the final state of the lair with every row on a separate line. On the last line, print either **"dead: {row} {col}"** or **"won: {row} {col}"**. Row and col are the coordinates of the cell where the player has died or the last cell he has been in before escaping the lair.

Input

- On the first line of input, the numbers **N** and **M** are received – the number of **rows** and **columns** in the lair
- On the next N lines, each row is received in the form of a string. The string will contain only ".", "B", "P". All strings will be the same length. There will be only one "P" for all the input
- On the last line, the directions are received in the form of a string, containing "R", "L", "U", "D"

Output

- On the first N lines, print the final state of the bunny lair
- On the last line, print the outcome – "won:" or "dead:" + {row} {col}

Constraints

- The dimensions of the lair are in range [3…20]
- The directions string length is in range [1…20]

Examples

Input	Output	Input	Output
5 8 ....B ...B.... ....B..B ....P..... ULLL	BBBBBBBB BBBBBBBB BBBBBBBB .BBBBBBB ..BBBBBB won: 3 0	4 5 .... .B... ...P. LLLLLLL	.B... BBB.. BBBB. BBB.. dead: 3 1