LAB: MULTIDIMENSIONAL ARRAYS

You can check your solutions in Judge

1. SUM MATRIX ELEMENTS

Write a program that **reads a matrix** from the console and prints:

- Count of **rows**
- Count of **columns**
- Sum of all **matrix elements**

On the first line, you will get matrix sizes in format **`[rows, columns]`**

EXAMPLES

Input	Output
3, 6 7, 1, 3, 3, 2, 1 1, 3, 9, 8, 5, 6 4, 6, 7, 9, 1, 0	3 6 76

HINTS

- On the next **`[rows]`** lines, you will get elements for each column separated with coma and whitespace.

2. SUM MATRIX COLUMNS

Create a program that **reads a matrix** from the console and prints the sum for each column. On the first line, you will get matrix **rows**. On the next **rows** lines, you will get elements for each column separated with a space.

EXAMPLES

Input	Output		
3, 6 7 1 3 3 2 1 1 3 9 8 5 6 4 6 7 9 1 0	12 10 19 20 8 7	3, 3 1 2 3 4 5 6 7 8 9	 12 15 18

HINTS

- Read matrix sizes.
- On the next lines, read the columns.
- Traverse the matrix and sum all elements in each column.
- Print the sum and continue with the other columns.

3. PRIMARY DIAGONAL

Create a program that finds the **sum of elements from the matrix's primary diagonal**.

	0	1	2
0	11	2	4
1	4	5	6
2	10	8	-12

primary diagonal
sum = 11 + 5 -12 = 4

INPUT

- On the **first line,** you are given the integer **N** – the size of the square matrix.
- The next **N lines,** hold the values for **every row** – **N** numbers separated by a space.

EXAMPLES

Input	Output		
3 11 2 4 4 5 6 10 8 -12	4	3 1 2 3 4 5 6 7 8 9	15

4. SYMBOL IN MATRIX

Create a program that reads **N**, a number representing **rows** and **cols** of a **matrix**. On the next **N** lines, you will receive rows of the matrix. Each row consists of ASCII characters. After that, you will receive a symbol. Find the **first occurrence** of that symbol in the matrix and print its position in the format: "**`({row}, {col})`**". If there is no such symbol, print an error message "**`{symbol} does not occur in the matrix`**".

EXAMPLES

Input	Output	Input	Output
3 ABC DEF X!@ !	(2, 1)	4 asdd xczc qwee qefw 4	4 does not occur in the matrix

5. SQUARE WITH MAXIMUM SUM

Create a program that **reads a matrix** from the console. Then find the biggest sum of the **2x2 submatrix** and print it to the console.
On the first line, you will get matrix sizes in format **`rows, columns.`**
On the next **`rows`** lines, you will get elements for each **column,** separated with a comma and a space.
Print the **biggest top-left** square, which you find, and the sum of its elements.

EXAMPLES

Input	Output		
3, 6	9 8 7 9 33	2, 4 10, 11, 12, 13	12 13 16 17 58

7, 1, 3, 3, 2, 1 1, 3, 9, 8, 5, 6 4, 6, 7, 9, 1, 0		14, 15, 16, 17	

HINTS

- Think about **`IndexOutOfRangeException()`.**
- If you find more than one max square, print the top-left one (min row, then min column).

6. JAGGED-ARRAY MODIFICATION

Create a program that **reads a matrix** from the console. On the first line, you will get matrix **`rows`**. On the next **`rows`** lines, you will get elements for each **column** separated with **space**. You will be receiving commands in the following format:

- **`Add {row} {col} {value}`** – **Increase** the number at the given **coordinates** with the **value.**
- **`Subtract {row} {col} {value}`** – **Decrease** the number at the given **coordinates** by the **value**.

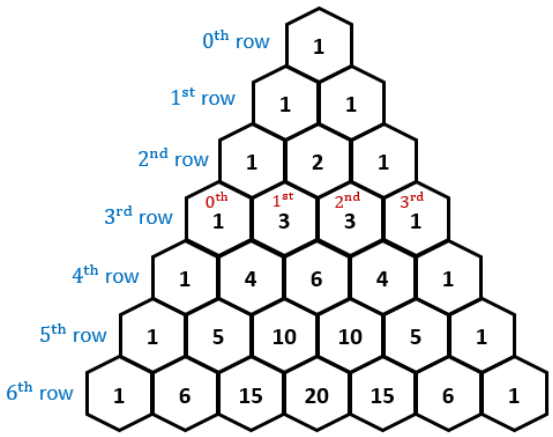
Coordinates might be invalid. In this case, you should print **`"Invalid coordinates"`**. When you receive **`"END"`** you should print the matrix and stop the program.

EXAMPLES

Input	Output	Input	Output
3 1 2 3 4 5 6 7 8 9 10 Add 0 0 5 Subtract 1 2 2 Subtract 1 4 7 END	Invalid coordinates 6 2 3 4 5 4 7 8 9 10	4 1 2 3 4 5 8 7 6 5 4 3 2 1 Add 4 4 100 Add 3 3 100 Subtract -1 -1 42 Subtract 0 0 42 END	Invalid coordinates Invalid coordinates -41 2 3 4 5 8 7 6 5 4 3 2 101

7. PASCAL TRIANGLE

The **Pascal's triangle** may be constructed in the following manner: in row `0` (the topmost row), there is a unique nonzero entry `1`. Each entry of each subsequent row is constructed by adding the number above and to the left with the number above and to the right, treating blank entries as `0`:



If you want more info about Pascal's triangle here.

Write a program to **print the Pascal's triangle** of given size **n**.

EXAMPLES

Input	Output	Input	Output
4	1 1 1 1 2 1 1 3 3 1	13	1 1 1 1 2 1 1 3 3 1 1 4 6 4 1 1 5 10 10 5 1

			1 6 15 20 15 6 1 1 7 21 35 35 21 7 1 1 8 28 56 70 56 28 8 1 1 9 36 84 126 126 84 36 9 1 1 10 45 120 210 252 210 120 45 10 1 1 11 55 165 330 462 462 330 165 55 11 1 1 12 66 220 495 792 924 792 495 220 66 12 1

HINTS

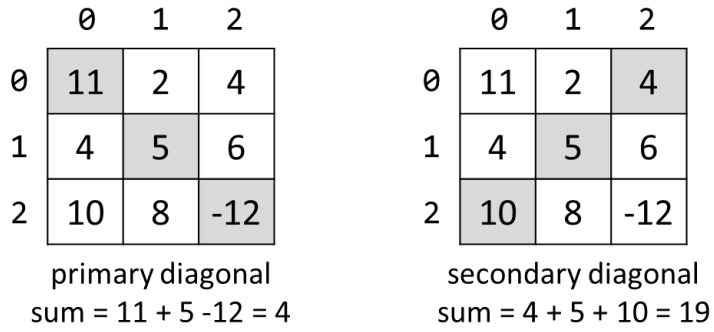
- The input number **n** will be **1 <= n <= 60**.
- Think about the proper **type** for elements in the array.
- Don't be scared to use **more and more arrays**.
- Use a **jagged array**, with triangular form.
- Each row is created by summing two elements from the previous rows.
- You may use

EXERCISE: MULTIDIMENSIONAL ARRAYS

- You can check your solutions in Judge
- Ask your questions here https://www.slido.com/ by entering the code **#csharp-advanced**

1. DIAGONAL DIFFERENCE

Create a program that finds the **difference between the sums of the square matrix diagonals** (absolute value).



INPUT

- On the **first line**, you are given the integer **N** – the size of the square matrix.
- The next **N lines**, hold the values for **every row** – **N** numbers separated by a space.

OUTPUT

- Print **the absolute** difference between **the sums** of the primary and the secondary diagonal.

HINT

- Use a **single** loop **`i`** = **`[1 … n]`** to sum the diagonals.
- The **primary diagonal** holds all cells **{`row`, `col`}** where **`row`** == **`col`** == **`i`**.
- The **secondary diagonal** holds all cells **{`row`, `col`}** where **`row`** == **`i`** and **`col`** == **`n-1-i`**.

EXAMPLES

Input	Output	Comments
3 11 2 4 4 5 6 10 8 -12	15	**Primary diagonal:** sum = 11 + 5 + (-12) = 4 **Secondary diagonal:** sum = 4 + 5 + 10 = 19 **Difference:** |4 - 19| = 15

2. SQUARES IN MATRIX

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

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 number of all the squares matrixes 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.

3. MAXIMAL SUM

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

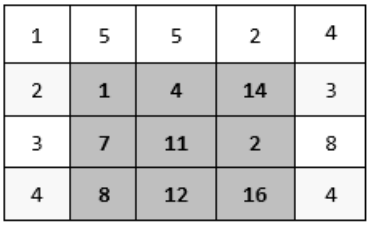
INPUT

- 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		Sum = 75 1 4 14 7 11 2 8 12 16

4. MATRIX SHUFFLING

Write a program that reads a string matrix from the console and performs certain operations with its elements. User input is provided in a similar way as in the problems above – first, you read the **dimensions** and then the **data**.

Your program should then receive commands in the format: "**`swap row1 col1 row2 col2`**" where `row1`, `col1`, `row2`, `col2` are **coordinates** in the matrix. For a command to be valid, it should start with the "**`swap`**" keyword along with **four valid coordinates** (no more, no less). You should **swap the values** at the given coordinates (cell `[row1, col1]` with cell `[row2, col2]`) **and print the matrix at each step** (thus you'll be able to check if the operation was performed correctly).

If the **command is not valid** (doesn't contain the keyword **`"swap"`**, has fewer or more coordinates entered or the given coordinates do not exist), print **`"Invalid input!"`** and move on to the next command. Your program should finish when the string **`"END"`** is entered.

EXAMPLES

Input	Output		

2 3 1 2 3 4 5 6 swap 0 0 1 1 swap 10 9 8 7 swap 0 1 1 0 END	5 2 3 4 1 6 Invalid input! 5 4 3 2 1 6	1 2 Hello World 0 0 0 1 swap 0 0 0 1 swap 0 1 0 0 END	Invalid input! World Hello Hello World

5. 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 N x M**. 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 to **fill the stairs**– 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 the 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

6. JAGGED ARRAY MANIPULATOR

Create a program that populates, analyzes and manipulates the elements of a matrix with an 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 an **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 an integer in the range [2…12].
- The **input** will always **follow** the **format above.**
- **Think about data types.**

EXAMPLES

Input	Output	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

7. 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** side 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	Input	Output	Input	Output

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

8. *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 with **every bomb**, one by one in the order they were given. When a bomb explodes deals damage **equal** to its **integer value**, to **all** the cells **around** it (in every direction and 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**. Afterward, 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 hold 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 cells in the format:
  `"Sum: {sumOfCells}"`
- At 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 the 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 neighbor 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	

9. *MINER

We get as input **the size** of the **field** in which our miner moves. The field is **always a square**. After that, we will receive the commands which represent the directions in which the miner should move. The miner **starts** from position – **'s'**. The commands will be: **`left`**, **`right`**, **up,** and **down**. If the miner has reached a side edge of the field and the next command indicates that he has to get out of the field, he must **remain in his current position and ignore the current command**. The possible characters that may appear on the screen are:

- ***** – a regular position on the field
- **e** – the end of the route.
- **c** - coal
- **s** - the place where the **miner starts**

Each time when the miner finds coal, he collects it and **replaces it with '*'**. Keep track of the **count of the collected coals**. If the miner collects all of the coals in the field, the program stops and you have to print the following message: **`"You collected all coals! ({rowIndex}, {colIndex})"`**.

If the miner **steps at 'e' the game is over (the program stops)** and you have to print the following message: **`"Game over! ({rowIndex}, {colIndex})".`**

If there are no more commands and none of the above cases had happened, you have to print the following message: **`"{remainingCoals} coals left. ({rowIndex}, {colIndex})"`**.

INPUT

- **Field size** – an integer number.
- **Commands to move** the miner – an array of strings separated by ' '.
- **The field: some of the following characters (*, e, c, s),** separated by whitespace (' ').

OUTPUT

- There are three types of output:
  - If all the coals have been collected, print the following output: **`"You collected all coals! ({rowIndex}, {colIndex})"`**.
  - If you have reached the end, you have to stop moving and print the following line: **`"Game over! ({rowIndex}, {colIndex})"`**.
  - If there are no more commands and none of the above cases had happened, you have to print the following message: **`"{totalCoals} coals left. ({rowIndex}, {colIndex})"`**.

CONSTRAINTS

- The **field size** will be a 32-bit integer in the range [0…2147483647].
- The field will always have only one's.
- Allowed working time for your program: 0.1 seconds.
- Allowed memory: 16 MB.

EXAMPLES

Input	Output
5 up right right up right * * * c * * * * e * * * c * * s * * c * * * c * *	Game over! (1, 3)
4 up right right right down * * * e * * c * * s * c * * * *	You collected all coals! (2, 3)
6 left left down right up left left down down down * * * * * *	3 coals left. (5, 0)

e * * * c * * * c s * * * * * * * * c * * * c * * * c * * *	

10. *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 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 the range [3…20].
- The directions string length is in the 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. LLLLLLLL	.B... BBB.. BBBB. BBB.. dead: 3 1