# Exercises: Multidimensional Lists

Problems for in-class lab for the [Python Advanced Course @SoftUni](https://softuni.bg/courses/python-advanced).

Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/1835>.

## Diagonal Difference

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



### Input

* On the **first line**, you will receive an integer **N** - the size of the square matrix
* The next N **lines** holds the values for **every row** - N numbers separated by a single space

### Output

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

### 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 |
| 4  -7 14 9 -20  3 4 9 21  -14 6 8 44  30 9 7 -14 | 34 |  |

## 2 X 2 Squares in Matrix

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

### Input

* On the **first line**, you will receive **the matrix's dimensions** in format **"{row} {column}"** **- [1, 20]**
* On the next **rows** you will receive **characters**, separated by a single space

### Output

* Print the count of all square matrices you have found

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 3 4  A B B D  E B B B  I J B B | 2 | Two 2x2 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 2x2 squares of equal cells exist. |

## Maximal Sum

Write a program that reads a **rectangular matrix's dimensions** and finds **the 3x3** square that **has maximal sum of its elements**.

### Input

* On the first line, you will receive the rows **N** and columns **M** in format **"{row} {column}"** – integers in range **[1, 20]**
* On the next **N lines** you will receive **each row with its columns - integers**, separated by a single space

### Output

* On the first line print **the maximum sum of the elements in the 3x3 square** in format **"Sum = {sum}"**
* On the next 3 lines print the rows of the **elements** in the 3x3 square, separated by a single space

### 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 |
| 5 6  1 0 4 3 1 1  1 3 1 3 0 4  6 4 1 2 5 6  2 2 1 5 4 1  3 3 3 6 0 5 |  | Sum = 34  2 5 6  5 4 1  6 0 5 |

## Matrix Shuffling

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

Your program should receive commands in format: "**swap {row1} {col1} {row2} {col2}**" where row1, row2, col1, col2 are **coordinates** in the matrix. A **valid** command **starts** with the "**swap**" keyword along with **four valid coordinates** (no more, no less).

* If the **command is valid**, you should **swap the values** at the given indexes **and print the matrix at each step** (thus you will be able to check if the operation was performed correctly).
* If the **command is not valid** (does not 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 |

## Snake Moves

You are tasked to visualize a snake's **zigzag path** in a **rectangular matrix with size N x M**.

The **snake** is represented by **a string**. It starts moving from the **top-left corner** to the **right**. When the snake reaches the end of the row, it slithers its way **down to the next row and turns left**. The moves are repeated to the very end. 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 path completely** - if you reach **the end** of the string representing the snake, start **again at the first symbol**. After you fill the matrix with the snake's path, you should print it.

### Input

The input data consists of exactly two lines:

* On the first line, you will receive the **dimensions N x M** of the field in format: **"{rows} {columns}"**.
* On the second line you will receive the string representing the **snake**

### Output

* You should print the **snake's zigzag path of size N x M** (rows x columns)

### 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 |

## 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**.

A chess knight has **8 possible moves** it can make, as illustrated. It can move to the **nearest** square but **not on the same**[**row**](https://en.wikipedia.org/wiki/Glossary_of_chess#rank), [**column**](https://en.wikipedia.org/wiki/Glossary_of_chess#file), or [**diagonal**](https://en.wikipedia.org/wiki/Glossary_of_chess#diagonal). (Foe example it can move two squares horizontally, then one square vertically, or it can move 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**.

You will receive a board with **"K"** for knights and "**0"** for empty cells. Your task is to **remove knights** until there are **no knights left that can attack one another**.

### Input

* On the first line, you will receive integer **N -** size of the board
* On the next **N** lines, you will receive strings with **"K"** and "**0"**

### Output

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

### Constraints

* The 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 |

## \*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 must detonate **every** **bomb**, in the order they were given. When a bomb explodes, it 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 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 **9** will explode and reduce its neighbour cells. In the end the bomb with coordinates **2,0** and value **9** will explode. After that you have to print the count of the alive cells, which are 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 |  |

## \*Miner

We receive **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 the edge of the field and the next command indicates that he has to get out of the field, he must **remain on his current possition 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**

When the miner finds a 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 … 2 147 483 647].**
* 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  \* \* \* \* \* \*  e \* \* \* c \*  \* \* c s \* \*  \* \* \* \* \* \*  c \* \* \* c \*  \* \* c \* \* \* | 3 coals left. (5, 0) |

## \*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 Python 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 consists **only** 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** every step, 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** |
| 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 |