# Exercises: Multidimensional Lists

Problems for in-class lab for the [Python Advanced Course @SoftUni](#).

Submit your solutions in the SoftUni judge system at [https://judge.softuni.org/Contests/1835.](https://judge.softuni.org/Contests/1835)

## 1. Diagonals

Using a **nested list comprehension,** write a program that reads **rows** of a **square matrix** and its **elements**, separated by a comma and a space **`", "`**. You should find the matrix's **diagonals**, prints them and their **sum** in the format:

**`"Primary diagonal: {element1}, {element2}, … {elementN}. Sum: {sum_of_primary}`**
**`Secondary diagonal: {element1}, {element2}, … {elementN}. Sum: {sum_of_secondary}"`**.

### Examples

| Input | Output |
|---|---|
| 3<br>1, 2, 3<br>4, 5, 6<br>7, 8, 9 | Primary diagonal: 1, 5, 9. Sum: 15<br>Secondary diagonal: 3, 5, 7. Sum: 15 |

## 2. Diagonal Difference

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

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

primary diagonal
sum = 11 + 5 -12 = 4

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

secondary diagonal
sum = 4 + 5 + 10 = 19

On the **first line**, you will receive an integer **N** - the size of a square matrix. The following **N lines** hold the values for **each column** - **N** numbers separated by a single space. Print **the absolute** difference between **the primary and the secondary diagonal sums**.

### Examples

| Input | Output | Comments |
|---|---|---|
| 3<br>11 2 4<br>4 5 6<br>10 8 -12 | 15 | Primary diagonal: sum = 11 + 5 + (-12) = 4<br>Secondary diagonal: sum = 4 + 5 + 10 = 19<br>Difference: \|4 - 19\| = 15 |
| 4<br>-7 14 9 -20<br>3 4 9 21 | 34 | |

| | | |
|---|---|---|
| `-14 6 8 44`<br>`30 9 7 -14` | | |

# 3. 2x2 Squares in Matrix

Find the number of all **2x2 squares containing identical chars** in a matrix. On the **first line**, you will receive **the matrix's dimensions** in the format **"{rows} {columns}"**. On the following **rows,** you will receive **characters** separated by a single space. Print the **number** of **all square matrices** you have found.

## Examples

| Input | Output | Comments |
|---|---|---|
| `3 4`<br>`A B B D`<br>`E B B B`<br>`I J B B` | `2` | Two 2x2 squares of equal cells:<br>`A B B D    A B B D`<br>`E B B B    E B B B`<br>`I J B B    I J B B` |
| `2 2`<br>`a b`<br>`c d` | `0` | No 2x2 squares of equal cells exist. |
| `5 4`<br>`A A B D`<br>`A A B B`<br>`I J B B`<br>`C C C G`<br>`C C K P` | `3` | Three 2x2 squares of equal cells:<br>`A A B D  A A B D  A A B D`<br>`A A B B  A A B B  A A B B`<br>`I J B B  I J B B  I J B B`<br>`C C C G  C C C G  C C C G`<br>`C C K P  C C K P  C C K P` |

# 4. Maximal Sum

Write a program that reads a **rectangular matrix's dimensions** and finds **the 3x3** square with a maximum **sum of its elements**. There will be **no case** with two or more **3x3** squares with **equal** maximal sum.

## Input

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

## Output

- On the first line, print **the maximum sum of the elements in the 3x3 square** in the format **"Sum = {sum}"**
- On the following 3 lines, **print each element of the found submatrix**, separated by a single space

## Examples

| Input | Matrix | Output |
|---|---|---|

<table>
<tr><td>4 5<br>1 5 5 2 4<br>2 1 4 14 3<br>3 7 11 2 8<br>4 8 12 16 4</td><td><table><tr><td>1</td><td>5</td><td>5</td><td>2</td><td>4</td></tr><tr><td>2</td><td>1</td><td>4</td><td>14</td><td>3</td></tr><tr><td>3</td><td>7</td><td>11</td><td>2</td><td>8</td></tr><tr><td>4</td><td>8</td><td>12</td><td>16</td><td>4</td></tr></table></td><td>Sum = 75<br>1 4 14<br>7 11 2<br>8 12 16</td></tr>
<tr><td>5 6<br>1 0 4 3 1 1<br>1 3 1 3 0 4<br>6 4 1 2 5 6<br>2 2 1 5 4 1<br>3 3 3 6 0 5</td><td></td><td>Sum = 34<br>2 5 6<br>5 4 1<br>6 0 5</td></tr>
</table>

# 5. Matrix of Palindromes

Write a program to generate the following **matrix of palindromes** of **3** letters with **r** rows and **c** columns like the one in the examples below.

- **Rows** define the **first** and the **last** letter: row 0 → 'a', row 1 → 'b', row 2 → 'c', …
- **Columns + rows** define the **middle** letter:
  - column 0, row 0 → 'a', column 1, row 0 → 'b', column 2, row 0 → 'c', …
  - column 0, row 1 → 'b', column 1, row 1 → 'c', column 2, row 1 → 'd', …

## Input

- The numbers **r** and **c** stay at the first line at the input in the format **`"{rows} {columns}"`**
- **r** and **c** are integers in the range **[1, 26]**

## Examples

| Input | Output |
|---|---|
| 4 6 | aaa aba aca ada aea afa<br>bbb bcb bdb beb bfb bgb<br>ccc cdc cec cfc cgc chc<br>ddd ded dfd dgd dhd did |
| 3 2 | aaa aba<br>bbb bcb<br>ccc cdc |

# 6. Matrix Shuffling

Write a program that reads a matrix from the console and performs certain operations with its elements. User input is provided similarly to the problems above - first, you read the **dimensions** and then the **data**.

Your program should receive commands in the format: **`"swap {row1} {col1} {row2} {col2}"`** where (**`"row1"`**, **`"col1"`**) and (**`"row2"`**, **`"col2"`**) are the **coordinates of two points** in the matrix. A **valid** command **starts** with the "**swap**" keyword along with **four valid coordinates** (no more, no less), separated by a single space.

- 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 are **not valid**), print **"Invalid input!"** and **move on** to the following command. **A negative value makes the coordinates not valid.**

Your program should finish when the command **"END"** is entered.

## Examples

| Input | Output |
|---|---|
| 2 3<br>1 2 3<br>4 5 6<br>swap 0 0 1 1<br>swap 10 9 8 7<br>swap 0 1 1 0<br>END | 5 2 3<br>4 1 6<br>Invalid input!<br>5 4 3<br>2 1 6 |
| 1 2<br>Hello World<br>0 0 0 1<br>swap 0 0 0 1<br>swap 0 1 0 0<br>END | Invalid input!<br>World Hello<br>Hello World |

# 7. Snake Moves

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

A string represents the snake. 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's path is as long as it takes to **fill the matrix completely** - if you reach **the end** of the string representing the snake, start **again at the first symbol**. In the end, you should **print the snake's path**.

## 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<br>SoftUni | SoftUn<br>UtfoSi<br>niSoft<br>foSinU<br>tUniSo |
| 1 4<br>Python | Pyth |

# 8. *Bombs

You will be given a square matrix of integers, each integer separated by a **single space**, and each row will be on a new line. On the last line of input, you will receive **indexes** - **coordinates** of 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 **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**. Afterward, **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 following **N** lines hold each column's values - **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: {alive_cells}**"
- On the second line, you need to print the sum of all alive cells in the format:
  "**Sum: {sum_of_cells}**"
- In the end, print the matrix. A space must separate the cells.

## 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<br>8 3 2 5<br>6 4 7 9<br>9 9 3 6<br>6 8 1 2<br>1,2 2,1 2,0 | Alive cells: 3<br>Sum: 12<br>8 -4 -5 -2<br>-3 -3 0 2<br>0 0 -4 -1<br>-3 -1 -1 2 | 1) The bomb with value 7 will explode and reduce the values of the cells around it.<br>2) The bomb with coordinates 2,1 and value 9 will explode and reduce its neighbor cells.<br>3) The bomb with coordinates 2,0 and value 9 will explode.<br>After that, you have to print the count of the alive cells - 3, and their sum - 12. Print the matrix after the explosions. |
| 3<br>7 8 4<br>3 1 5<br>6 4 9<br>0,2 1,0 2,2 | Alive cells: 3<br>Sum: 8<br>4 1 0<br>0 -3 -8<br>3 -8 0 | |

# 9. *Miner

You are going to create a game called "Miner".

First, you will receive the **size** of a **square field** in which the miner should move.

On the second line, you will receive the **commands** for the miner's movement, separated by a single space. The possible commands are **"left"**, **"right"**, **"up"** and **"down"**.

In the end, you will receive **each row of the field** on a separate line. 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** - miner

The **miner starts moving** from the position **"s"**. He should perform the given commands **successively**, moving with **only one position** in the given direction. If the miner has reached the **edge of the field** and the following command indicates that he has to get out of the area, he must **remain in his current position** and **ignore** the command.

When the miner finds **coal**, he **collects it** and **replaces** it with **"*"**. Keep track of the collected coal. In the end, you should print whether the miner has **succeeded in collecting the coal** or not and his **final position**:

- If the miner has **collected all coal in the field**, the program stops, and you should **print** the message: "**You collected all coal! ({row_index}, {col_index})**".
- If the miner **steps at "e"**, the game is **over** (the program stops), and you should **print** the message: **"Game over! ({row_index}, {col_index})"**.
- If there are **no more commands** and **none** of the above cases had **happened**, you should **print** the message: **"{number_of_remaining_coal} pieces of coal left. ({row_index}, {col_index})"**.

## Input

- **Field size** - an integer number
- **Commands to move** the miner - a sequence of directions, separated by single whitespace (" ")
- **The field: some of the following characters ("*", "e", "c ", "s"),** separated by a single whitespace (" ")

## Output

- There are three types of output as mentioned above.

## 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"**

## Examples

| Input | Output |
|---|---|
| 5<br>up right right up right<br>* * * c *<br>* * * e *<br>* * c * *<br>s * * c *<br>* * c * * | Game over! (1, 3) |
| 4<br>up right right right down<br>* * * e<br>* * c *<br>* s * c<br>* * * * | You collected all coal! (2, 3) |
| 6<br>left left down right up left left down down down<br>* * * * * *<br>e * * * c *<br>* * c s * *<br>* * * * * *<br>c * * * c *<br>* * c * * * | 3 pieces of coal left. (5, 0) |

## 10. *Radioactive Mutant Vampire Bunnies

*You come across an old JS Basics teamwork game. It is about bunnies that multiply extremely fast. There's also a player that should escape from their lair. You like the game, so you decide to port it to Python because that's your language of choice. The last thing left is the algorithm that determines if the player will escape the lair or not.*

First, you will receive a line holding integers **N** and **M**, representing the lair's rows and columns.

Next, you receive **N** strings that can consist **only** of **"."**, **"B"**, **"P"**. They represent the initial state of the lair. There will be **only** one player. The **bunnies** are marked with **"B",** the **player** is marked with **"P"**, and **everything** else is free space, marked with a dot **"."**.

Then you will receive a string with **commands** (e.g., **LRRULUD**) - each letter represents the **next move** of the player:

- **L** - the player should move one position to the left

- **R** - the player should move one position to the right
- **U** - the player should move one position up
- **D** - the player should move one position down

**After** every step made, each bunny spreads **one position up**, **down**, **left**, and **right**. If the player **moves** to a bunny cell or a bunny **reaches** the player, the player dies. If the player goes **out** of the lair **without** encountering a bunny, the player wins.

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 cases** where the moves of the player **end before he dies or escapes**.

In the end, **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 cell coordinates 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 following 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:
  - If the player won - **`"won: {row} {col}"`**
  - If the player dies - **`"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 | Input | Output |
|---|---|---|---|---|---|
| 5 6<br>.....P<br>......<br>...B..<br>......<br>......<br>ULDDDR | ......<br>...B..<br>..BBB.<br>...B..<br>......<br>won: 0 5 | 4 5<br>.....<br>.....<br>.B...<br>...P.<br>LLLLLLLL | .B...<br>BBB..<br>BBBB.<br>BBB..<br>dead: 3 1 | 5 8<br>.......B<br>...B....<br>....B..B<br>........<br>..P.....<br>ULLL | BBBBBBBB<br>BBBBBBBB<br>BBBBBBBB<br>.BBBBBBB<br>..BBBBBB<br>won: 3 0 |