# Exercises: Lists and Matrices

Problems for exercises and homework for the [“Programming Fundamentals” course @ SoftUni](https://softuni.bg/courses/programming-fundamentals).

You can check your solutions here: <https://judge.softuni.bg/Contests/208/Lists-and-Matrices-Exercises>

## Max Sequence of Equal Elements

Read a **list of integers** and find the **longest sequence of equal elements**. If several exist, print the **leftmost**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3 4 4 **5 5 5** 2 2 | 5 5 5 |
| **7 7** 4 4 5 5 3 3 | 7 7 |
| 1 2 **3 3** | 3 3 |

### Hints

* Scan positions **p** from left to right and keep the **start** and **length** of the current sequence of equal numbers ending at **p**.
* Keep also the currently best (longest) sequence (bestStart + bestLength) and update it after each step.

## Matrix of Palindromes

Write a program to generate the following **matrix of palindromes** of **3** letters with **r** rows and **c** columns like at 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 number r and c stay at the first line at the input.
* Constraints: r and c are integers in the range [1…26]; r + c ≤ 27.

### Examples

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

### Hints

* Use two nested loops to generate the matrix.
* Print the matrix row by row in a loop.

## Diagonal Difference

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



### Input

* The **first line** holds a number n – the size of matrix.
* The next n **lines** hold the **values for every row** – n numbers separated by a space.

### 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 |

### Hints

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.

## 2x2 Squares in Matrix

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

* The matrix size is given at the first row (**rows** and **columns**).
* Matrix characters come at the next **rows** lines (space separated).

### 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. |

### Hints

Check all possible 2 x 2 squares for equal elements.

## Max Platform 3 x 3

Write a program, which reads a rectangular matrix of integers of size of r rows by c columns. Find in the matrix a **platform of size 3 x 3** **with a maximal sum**.

### Input

* The first line holds the number of rows r and the number of columns c.
* The next r lines hold the elements of the matrix.
* Constraints: 3 ≤ r, c ≤ 1000.

### Output

* At the **first line** of the output **print the sum**.
* At the next line **print the platform itself**.
* If **several** platforms of equal sum exist, print the one that is located in the **most upper-left** position.

### Examples

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Input** | **Output** |  | **Input** | **Output** |  | **Input** | **Output** |
| 4 4  5 **6 2 8**  3 **1 9 5**  8 **1 6 9**  1 5 3 4 | 47  6 2 8  1 9 5  1 6 9 |  | 5 6  1 2 4 8 9 6  2 4 1 3 4 2  2 7 **9 9 9** 7  8 6 **9 9 9** 6  9 5 **9 9 9** 9 | 81  9 9 9  9 9 9  9 9 9 | 4 6  **1 1 1** 1 1 1  **1 1 1** 1 1 1  **1 1 2** 1 1 1  1 1 1 1 1 1 | 10  1 1 1  1 1 1  1 1 2 |

### Hints

Check all possible 3 x 3 squares.

## Hourglass Sum

You are given a 2D array (matrix) of size 6 x 6. An **hourglass** in an array is defined as a set of cells shaped like this:

|  |  |  |
| --- | --- | --- |
| **a** | **b** | **c** |
|  | **d** |  |
| **e** | **f** | **g** |

For example, if we create an hourglass using the number 1 within an array full of zeros, it may look like this:

1 1 1 0 0 0

0 1 0 0 0 0

1 1 1 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0

Actually, there are many hourglasses in the array above. The three topmost hourglasses are the following:

1 1 1 1 1 0 1 0 0

1 0 0

1 1 1 1 1 0 1 0 0

The **sum of an hourglass** is the sum of all the numbers within it. The sum for the hourglasses above are 7, 4, and 2, respectively.

Write a program to calculate and print the **largest sum among all the hourglasses** in the matrix.

### Input

* The matrix is given as 6 text lines, each holding 6 numbers, separated by space.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 1 1 0 0 0  0 1 0 0 0 0  1 1 1 0 0 0  0 0 2 4 4 0  0 0 0 2 0 0  0 0 1 2 4 0 | 19 |

### Hints

Check all possible hourglass positions.

## \*\* Matrix Generator

Write a program, which creates **matrices** like those in the **figures below** and prints them formatted at the console. The **type** and the **size** of the matrices will be read from the console. E.g. matrices with size of 4 x 4:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Type A** | | | |  | **Type B** | | | |  | **Type C \*** | | | |  | **Type D \*\*** | | | |
| 1 | 5 | 9 | 13 |  | 1 | 8 | 9 | 16 |  | 7 | 11 | 14 | 16 |  | 1 | 12 | 11 | 10 |
| 2 | 6 | 10 | 14 |  | 2 | 7 | 10 | 15 |  | 4 | 8 | 12 | 15 |  | 2 | 13 | 16 | 9 |
| 3 | 7 | 11 | 15 |  | 3 | 6 | 11 | 14 |  | 2 | 5 | 9 | 13 |  | 3 | 14 | 15 | 8 |
| 4 | 8 | 12 | 16 |  | 4 | 5 | 12 | 13 |  | 1 | 3 | 6 | 10 |  | 4 | 5 | 6 | 7 |

### Examples

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input** | **Output** |  | **Input** | **Output** |  | **Input** | **Output** |  | **Input** | **Output** |
| A 4 4 | 1 5 9 13  2 6 10 14  3 7 11 15  4 8 12 16 |  | B 3 2 | 1 6  2 5  3 4 |  | C 4 5 | 7 11 15 18 20  4 8 12 16 19  2 5 9 13 17  1 3 6 10 14 |  | D 3 4 | 1 10 9 8  2 11 12 7  3 4 5 6 |

### Hints

* **Type A**: start from {row = 0, col = 0, value = 1}. Move down (row++; value++). When the bottom of the matrix is reached (row == rowsCount), move to the next column and the top row (col++; row = 0). Stop when the last column is passed (col == colCount).
* **Type B**: start from {row = 0, col = 0, dir = 1 (**down**), value = 1}. Move in the current direction (up or down): row += dir; value++. When the top of the bottom of the matrix is reached (row == rowsCount or row == -1), move to the next column and change the direction (col++; dir = -dir). Stop when the last column is passed (col == colCount).
* **Type C**: fill the matrix as series of diagonal lines:
  + Initially start with value == 1.
  + Start with startRow = rowCount-1 and decrease startRow in a loop until the entire matrix is filled (value > rowsCount \* colsCount).
    - At each step start from {row = startRow; col = 0} and move down-right in a loop (row++; col++) until the bottom of the matrix is passed (row == rowsCount).
    - In the inner loop, when {row, col} is a valid cell in the matrix, fill the next value in the cell (matrix[row, col] = value++).
  + Finally, print the filled matrix.
* **Type D**: move consequently down, right, up, left, again down, again right, again up, again left, etc. After each move fill a matrix cell with the next value. Start from {row = 0; col = 0; value = 0}.
  + Move rowsCount times **down**.
  + Move colsCount-1 times **right**.
  + Move rowsCount-1 times **up**.
  + Move colsCount-2 times **left**.
  + Move rowsCount-2 times **down**.
  + Move colsCount-3 times **right**.
  + Move rowsCount-3 times **up**.
  + …
  + **Stop** when the **number of times** for some direction becomes 0.

## \* Largest Frame in Matrix

Find the **largest rectangular frame** of equal numbers in a **matrix of integers**.

* If several equal sized largest frames exist, **print all of them**, **sorted alphabetically** (as strings).
* Frames could span one or several rows and one or several columns.
* The numbers at the frame borders should be the same.
* Frames are allowed to **overlap**.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 3 7  3 5 5 5 3 3 3  3 5 0 5 3 0 3  3 5 5 5 3 3 3 | 3x3, 3x3 | Many frames exist in the matrix: 1x1, 2x1, 1x2, 3x1, 1x3, 3x3, … The largest frame has size 3x3. There are two frames of the same largest size of 9 cells (shown in different colors on the left), both of size 3 x 3. |
| 3 6  7 7 0 8 8 8  7 7 0 8 8 8  7 7 7 7 7 7 | 1x6, 2x3, 3x2 | Many frames exist in the matrix: 1x1, 2x1, 1x2, 3x1, 1x3, 2x3, 3x2, 3x3, … The largest frame has size 6. There are 3 frames of the same largest size of 6 cells (shown in different colors on the left): 2x3, 3x2, and 1x6. After sorting (as text) the result is: 1x6, 2x3, 3x2. |

## \*\* Largest Increasing Subsequence (LIS)

Read a **list of integers** and find the **longest increasing subsequence** (LIS). If several such exist, print the **leftmost**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| **1** | 1 |
| 7 **3 5** 8 -1 0 **6 7** | 3 5 6 7 |
| **1 2** 5 **3 5** 2 4 1 | 1 2 3 5 |
| **0** 10 20 30 30 40 **1** 50 **2 3 4 5 6** | 0 1 2 3 4 5 6 |
| 11 12 13 **3** 14 **4** 15 **5 6 7 8** 7 **16** 9 8 | 3 4 5 6 7 816 |
| **3** 14 **5** 12 15 **7 8 9 11** 10 1 | 3 5 7 8 9 11 |

### Hints

* Assume we have n numbers in an array nums[0…n-1].
* Let len[p] holds the length of the longest increasing subsequence (LIS) ending at position p.
* In a for loop, we calculate shall len[p] for p = 0 … n-1 as follows:
  + Let left is the leftmost position on the left of p (left < p), such that len[left] is the maximal possible.
  + Then, len[p] = 1 + len[left]. If left does not exist, len[p] = 1.
  + Also save prev[p] = left (we hold if prev[] the previous position, used to obtain the best length for position p).
* Once the values for len[0…n-1] are calculated, restore the LIS starting from position p such that len[p] is maximal and go back and back through p = prev[p].
* The table below illustrates these computations:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| nums[] | **3** | **14** | **5** | **12** | **15** | **7** | **8** | **9** | **11** | **10** | **1** |
| len[] | 1 | 2 | 2 | 3 | 4 | 3 | 4 | 5 | 6 | 6 | 1 |
| prev[] | -1 | 0 | 0 | 2 | 3 | 2 | 5 | 6 | 7 | 7 | -1 |
| LIS | {3} | {3,14} | {3,5} | {3,5,12} | {3,5,12,15} | {3,5,7} | {3,5,7,8} | {3,5,7,8,9} | {3,5,7,8,9,11} | {3,5,7,8,9,10} | {1} |

## \* Array Manipulator

Write a program that **reads an array of integers** from the console and **set of commands** and **executes them over the array**. The commands are as follows:

* **add <index> <element>** – adds element at the specified index (elements right from this position inclusively are shifted to the right).
* **addMany <index> <element 1> <element 2> … <element n>** – adds a set of elements at the specified index.
* **contains <element>** – prints the index of the first occurrence of the specified element (if exists) in the array or **-1** if the element is not found.
* **remove <index>** – removes the element at the specified index.
* **shift <positions>** – **shifts every element** of the array the number of positions **to the** **left** (with rotation).
  + For example, [1, 2, 3, 4, 5] -> shift 2 -> [3, 4, 5, 1, 2]
* **sumPairs** – sums the elements in the array by pairs (first + second, third + fourth, …).
  + For example, [1, 2, 4, 5, 6, 7, 8] -> [3, 9, 13, 8].
* **print** – stop receiving more commands and print the last state of the array.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 2 4 5 6 7  add 1 8  contains 1  contains -3  print | 0  -1  [1, 8, 2, 4, 5, 6, 7] |
| 1 2 3 4 5  addMany 5 9 8 7 6 5  contains 15  remove 3  shift 1  print | -1  [2, 3, 5, 9, 8, 7, 6, 5, 1] |
| 2 2 4 2 4  add 1 4  sumPairs  print | [6, 6, 6] |
| 1 2 1 2 1 2 1 2 1 2 1 2  sumPairs  sumPairs  addMany 0 -1 -2 -3  print | [-1, -2, -3, 6, 6, 6] |

## Sum Reversed Numbers

Write a program that reads sequence of numbers, reverses their digits, and prints their sum.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 123 234 12 | 774 | 321 + 432 + 21 = 774 |
| 12 12 34 84 66 12 | 220 | 21 + 21 + 43+ 48 + 66 + 21 = 220 |
| 120 1200 12000 | 63 | 21 + 21 + 21 = 63 |

## Bomb Numbers

Write a program that **reads sequence of numbers** and **special bomb number** with a certain **power**. Your task is to **detonate every occurrence of the special bomb number** and according to its power **his neighbors from left and right**. Detonations are performed from left to right and all detonated numbers disappear. Finally print the **sum of the remaining elements** in the sequence.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 1 2 2 4 2 2 2 9  4 2 | 12 | Special number is **4** with power 2. After detontaion we left with the sequence [1, 2, 9] with sum 12. |
| 1 4 4 2 8 9 1  9 3 | 5 | Special number is **9** with power 3. After detontaion we left with the sequence [1, 4] with sum 5. Since the 9 has only 1 neighbour from the right we remove just it (one number instead of 3). |
| 1 7 7 1 2 3  7 1 | 6 | Detonations are performed from left to right. We could not detonate the second occurance of 7 because its already destroyed by the first occurance. The numbers [1, 2, 3] survive. Their sum is 6. |
| 1 1 2 1 1 1 2 1 1 1  2 1 | 4 | The red and yellow numbers disappear in two sequential detonations. The result is the sequence [1, 1, 1, 1]. Sum = 4. |

## Tour

You are given a square matrix representing **distances between cities**. City names are numbers from 0 to N. Also you are a given a sequence of cities that needs to be visited in that order. You start from city 0. Your task is to write a program that reads the distances matrix, the sequence of numbers representing the numbers of the cities that will be visited starting from city 0 and finally print the total distance of the route. For example:

|  |  |  |  |
| --- | --- | --- | --- |
| **City** | **0** | **1** | **2** |
| **0** | 0 | 10 | 20 |
| **1** | 10 | 0 | 55 |
| **2** | 20 | 55 | 0 |

Distances between city 0 and city 1 is 10, between city 0 and city 2 is 20 and between 1 and 2 is 55.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 3  0 10 20  10 0 55  20 55 0  1 0 2 1 | 95 | 3 is the size of the matrix (3 rows and 3 cols)  Starting from city 0. The route is as follows:  0 -> 1 -> 0 -> 2 -> 1  0 -> 1 (distance 10)  1 -> 0 (distance 10)  0 -> 2 (distance 20)  2 -> 1 (distance 55)  Total distance: 95 |