# horizontal lineData Structures

Homework Assignment 4 - Dynamic Arrays

Problem 1 - Connected Components - 50 Points

Problem 2 - Matrix Class - 50 Points

**Notes and Requirements**

* Your submission must be your effort. You can not copy other students' code.
* This worksheet is graded on performance; Implementations must be correct.
* You are encouraged to visit our office hours to ask coding questions.
* Only the latest (most recent) submission is graded.
* Late submissions are not considered for grading.
* You can not use any third-party libraries.

**Some assignments on this worksheet are manually graded.**

## Problem 1 - Connected Components - 50 Points

A voxel grid is a three-dimensional array of voxels, where a voxel (short for "volumetric pixel") is the three-dimensional analog of a pixel in a 2D image. Each voxel represents a small volume element in space and is often associated with some physical property, such as density, temperature, or chemical composition. Voxel grids are commonly used in medical imaging, scientific visualization, and computer graphics to represent three-dimensional objects or scenes.

A connected component is a group of adjacent voxels that are connected to each other in the 3D space. Two adjacent voxels are connected if they share a common face, edge, or corner (i.e., they have the same x, y, or z coordinate and differ by one in one of the other two coordinates). Any set of connected voxels that are not connected to another set of connected voxels is considered a connected component.

| Voxel grid with two connected components | Single voxel with highlights |
| --- | --- |
|  |  |

The picture on the left shows two connected components of 10 voxels each. The components are placed in a 5x5x5 voxel grid. The image on the right shows a connected component represented by a single voxel. Highlighted in red are the voxel`s edges. Highlighted in green are the voxel`s corners. The blue areas show the voxel`s faces.

We are simplifying the definition of connected components so that voxels are explicitly considered connected when voxels share faces. Voxels are regarded as not connected when edges or corners are shared. Please see the following table for details and a visual representation:

| Connected Component | Not-Connected Component | Not-Connected Component |
| --- | --- | --- |
|  |  |  |
| the same face is shared  (picture shows 1 component  consisting of 2 voxels) | the same edge is shared  (picture shows 2 components  consisting out of 1 voxel each) | the same corner is shared (picture shows 2 components consisting out of 1 voxel each) |

A voxel grid can be represented by a multidimensional array in Python, as visualized in the following example. The example shows two connected components consisting of 3 voxels each. Connected component 1 is marked **red**, whereas connected component 2 is marked **blue**:

| 2x4x4 Array | 2x4x4 Voxel Grid | Notes |
| --- | --- | --- |
| voxel\_grid = [  [  [**1**, **1**, 0, 0],  [**1**, 0, 0, 0],  [0, 0, 0, 0],  [0, 0, 0, 0],  ],[  [0, 0, 0, 0],  [0, 0, 0, 0],  [0, 0, 0, **1**],  [0, 0, **1**, **1**],  ]  ] |  | The voxel grid has a dimension of 2x4x4 and has a total of 2 components.  - Component 1 has 3 voxels.  - Component 2 has 3 voxels |

**Your task**

Implement the recursive function *count\_components(voxel\_grid)* that returns the exact number of connected components in a given voxel grid. Your solution has to be recursive.

**Hints**

* You can use the provided *visualize* function to render and display a given voxel grid. Please comment this line of code when submitting your solution to Gradescope.
* The solution to this problem is unique. It is not possible that the number of connected components for a given voxel gird can be represented by two different values.
* The maximum number of dimensions for a voxel gird is (3) **n\*m\*k**, where n, m, k are any positive integer. There are no multidimensional array test cases exceeding this limit.
* You can copy the given examples into your IDE and create any custom voxel grid to test and verify the behavior of your function. You can find additional examples in the provided code.
* As per our definition, voxels attached via edges and corners are not considered connected.

**Requirements**

* Your function has to return the number of connected components of a given grid.
* Your solution has to utilize recursion to discover the voxel grid.
* You can not use global variables for storing any information.
* You can not simplify or modify the provided voxel grids.

**Examples**

| **Example 1** |  |  |
| --- | --- | --- |
| 1x1x1 Array | 1x1x1 Voxel Grid | Notes |
| voxel\_grid = [  [  [**1**]  ]  ] |  | The voxel grid has a dimension of 1x1x1 and has a total of 1 component.  - Component 1 has 1 voxel. |

| **Example 2** |  |  |
| --- | --- | --- |
| 5x5x5 Array | 5x5x5 Voxel Grid | Notes |
| voxel\_grid = [  [  [**1**, **1**, 0, 0, 0],  [**1**, **1**, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0]  ],[  [**1**, **1**, 0, 0, 0],  [**1**, **1**, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0]  ],[  [**1**, 0, 0, 0, 0],  [**1**, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, **1**],  [0, 0, 0, 0, **1**]  ],[  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, **1**, **1**],  [0, 0, 0, **1**, **1**]  ],[  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, **1**, **1**],  [0, 0, 0, **1**, **1**]  ]  ] |  | The voxel grid has a dimension of 5x5x5 and has a total of 2 components.  - Component 1 has 10 voxels.  - Component 2 has 10 voxels. |

| **Example 3** |  |  |
| --- | --- | --- |
| 2x1x2 Array | 2x1x2 Voxel Grid | Notes |
| voxel\_grid = [  [  [**1**, 0],  ],[  [0, **1**],  ]  ] |  | The voxel grid has a dimension of 2x1x2 and has a total of 2 component.  - Component 1 has 1 voxel.  - Component 2 has 1 voxel. |

| **Example 4** |  |  |
| --- | --- | --- |
| 3x3x3 Array | 3x3x3 Voxel Grid | Notes |
| voxel\_grid = [  [  [**1**, 0, 0],  [**1**, 0, 0],  [**1**, 0, 0],  ],[  [0, 0, 0],  [0, 0, 0],  [0, 0, 0],  ],[  [0, 0, 0],  [0, 0, 0],  [**1**, **1**, **1**],  ]  ] |  | The voxel grid has a dimension of 3x3x3 and has a total of 2 component.  - Component 1 has 3 voxels.  - Component 2 has 3 voxels. |

| **Example 5** |  |  |
| --- | --- | --- |
| 3x3x3 Array | 3x3x3 Voxel Grid | Notes |
| voxel\_grid = [  [  [**1**, 0, 0],  [**1**, 0, 0],  [**1**, 0, 0],  ],[  [**1**, 0, 0],  [0, **1**, 0],  [0, 0, **1**],  ],[  [0, 0, 0],  [0, 0, 0],  [**1**, **1**, **1**],  ]  ] |  | The voxel grid has a dimension of 3x3x3 and has a total of 3 components.  - Component 1 has 4 voxels.  - Component 2 has 1 voxel.  - Component 3 has 4 voxels. |

| **Example 6** |  |  |
| --- | --- | --- |
| 6x5x5 Array | 6x5x5 Voxel Grid | Notes |
| voxel\_grid = [  [  [**1**, **1**, **1**, **1**, **1**],  [**1**, 0, 0, 0, **1**],  [**1**, 0, 0, 0, **1**],  [**1**, 0, 0, 0, **1**],  [**1**, **1**, **1**, **1**, **1**],  ],[  [**1**, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [**1**, 0, 0, 0, **1**],  ],[  [**1**, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [**1**, 0, 0, 0, **1**],  ],[  [**1**, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [**1**, 0, 0, 0, **1**],  ],[  [**1**, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [**1**, 0, 0, 0, **1**],  ],[  [**1**, **1**, 0, 0, 0],  [**1**, 0, 0, 0, 0],  [**1**, 0, 0, 0, 0],  [**1**, 0, 0, 0, 0],  [**1**, **1**, **1**, **1**, **1**],  ]  ] |  | The voxel grid has a dimension of 6x5x5 and has a total of 4 components.  - Component 1 has 38 voxels. |

## Problem 2 - Matrix Class - 50 Points

Implement the class Matrix as outlined by the following UML diagram and provide the correct functionality for performing the listed matrix operations. Your implementation has to raise a ValueError exception if an operation cannot be performed.

| <<Class>>  Matrix |
| --- |
| - field: \_rows  - field: \_cols  - field: data |
| + constructor: \_\_init\_\_(rows, cols)  + method: \_\_add\_\_(other)  + method: \_\_sub\_\_(other)  + method: \_\_mul\_\_(other)  + method: \_\_truediv\_\_(other)  + method: \_\_eq\_\_(other)  + method: \_\_str\_\_()  + method: transpose()  + method: determinant()  + method: rank()  + method: scale(scalar)  + method: removeRow(row\_index)  + method: removeCol(col\_index) |

**Requirements**

* The transpose function has to work in place recursively.
* Exceptions have to be raised in case operations cannot be performed.
* Object equality comparisons have to be restricted to objects of the same type.
* You can not use the Python math library.
* Floating point numbers are rounded to 1 decimal digit, e.g., 1.0, 2.7, 5.8

**Hints**

* Multiplication and division are only possible with compatible dimensions.
* Calculating the determinant is only possible with compatible dimensions.
* Raise an error if dimensions are incompatible or removing column/row is invalid.

**Examples**

| **Addition (+)** |  |
| --- | --- |
| Square Matrix | Non-Square Matrix |
| matrix1 = Matrix(2, 2)  matrix1.data = [[1, 1], [1, 1]]  matrix2 = Matrix(2, 2)  matrix2.data = [[1, 1], [1, 1]]  (matrix1 + matrix2).data == [[2, 2], [2, 2]] | matrix1 = Matrix(2, 3)  matrix1.data = [[1, 1, 1], [1, 1, 1]]  matrix2 = Matrix(2, 3)  matrix2.data = [[1, 1, 1], [1, 1, 1]]  (matrix1 + matrix2).data == [[2, 2, 2], [2, 2, 2]] |

| **Subtraction (-)** |  |
| --- | --- |
| Square Matrix | Non-Square Matrix |
| matrix1 = Matrix(2, 2)  matrix1.data = [[1, 1], [1, 1]]  matrix2 = Matrix(2, 2)  matrix2.data = [[1, 1], [1, 1]]  (matrix1 - matrix2).data == [[0, 0], [0, 0]] | matrix1 = Matrix(2, 3)  matrix1.data = [[1, 1, 1], [1, 1, 1]]  matrix2 = Matrix(2, 3)  matrix2.data = [[1, 1, 1], [1, 1, 1]]  (matrix1 - matrix2).data == [[0, 0, 0], [0, 0, 0]] |

| **Multiplication (\*)** |  |
| --- | --- |
| Square Matrix | Non-Square Matrix |
|  | matrix1 = Matrix(2, 3)  matrix1.data = [[1, 2, 3], [4, 5, 6]]  matrix2 = Matrix(3, 2)  matrix2.data = [[1, 2], [3, 4], [5, 6]]  (matrix1 \* matrix2).data == [[22, 28], [49, 64]] |

| **Division(/)** |  |
| --- | --- |
| Square Matrix | Non-Square Matrix |
| matrix1 = Matrix(2, 2)  matrix1.data = [[1, 2], [4, 5]]  matrix2 = Matrix(2, 2)  matrix1.data = [[1, 2], [4, 5]]  (matrix1 / matrix2).data == [[1.0, 1.0], [1.0, 1.0]] | matrix1 = Matrix(2, 3)  matrix1.data = [[1, 2, 3], [4, 5, 6]]  matrix2 = Matrix(2, 3)  matrix1.data = [[1, 2, 3], [4, 5, 6]]  (matrix1 / matrix2).data == [[1.0, 1.0, 1.0], [1.0, 1.0, 1.0]] |
| **Equality (==)** |  |
| Square Matrix | Non-Square Matrix |
| matrix1 = Matrix(2, 2)  matrix1.data = [[1, 1], [1, 1]]  matrix1 == matrix1 # is true | matrix1 = Matrix(2, 3)  matrix1.data = [[1, 1, 1], [1, 1, 1]]  matrix1 == matrix1 # is true |

| **Stringification (print)** |  |
| --- | --- |
| Square Matrix | Non-Square Matrix |
| matrix1 = Matrix(2, 2)  matrix1.data = [[1, 1], [1, 1]]  matrix1.\_\_str\_\_() == "1 1\n1 1" | matrix1 = Matrix(2, 2)  matrix1.data = [[1, 1, 1], [1, 1, 1]]  matrix1.\_\_str\_\_() == "1 1 1\n1 1 1" |

| **Transpose (m^T)** |  |
| --- | --- |
| Square Matrix | Non-Square Matrix |
| matrix2 = Matrix(2, 2)  matrix2.data = [[-1, -2], [-3, -4]]  matrix2.transpose()  matrix2.data == [[-1, -3], [-2, -4]] | matrix1 = Matrix(2, 3)  matrix1.data = [[1, 2, 3], [4, 5, 6]]  matrix1.transpose()  matrix1.data == [[1, 4], [2, 5], [3, 6]] |

| **Determinant** |  |
| --- | --- |
| Square Matrix | Non-Square Matrix |
| matrix1 = Matrix(2, 2)  matrix1.data = [[1, 2], [3, 4]]  matrix1.determinant() == -2 |  |

| **Rank** |  |
| --- | --- |
| Square Matrix | Non-Square Matrix |
| matrix1 = Matrix(2, 2)  matrix1.data = [[1, 1], [1, 1]]  matrix1.rank() == 1 | matrix2 = Matrix(4, 4)  matrix2.data = [[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]  matrix2.rank() == 4 |

| **Scale** |  |
| --- | --- |
| Square Matrix | Non-Square Matrix |
| matrix1 = Matrix(2, 2)  matrix1.data = [[1, 1], [1, 1]]  matrix1.scale(2)  matrix1.data == [[2, 2], [2, 2]] | matrix2 = Matrix(2, 3)  matrix2.data = [[-1, -1, -1], [-1, -1, -1]]  matrix2.scale(2)  matrix2.data == [[-2, -2, -2], [-2, -2, -2]] |

| **RemoveRow** |  |
| --- | --- |
| Square Matrix | Non-Square Matrix |
| matrix1 = Matrix(2, 2)  matrix1.data = [[1, 2], [4, 5]]  matrix1.removeRow(1)  matrix1.data == [[1, 2]] | matrix1 = Matrix(2, 3)  matrix1.data = [[1, 2, 3], [4, 5, 6]]  matrix1.removeRow(1)  matrix1.data == [[1, 2, 3]] |

| **RemoveCol** |  |
| --- | --- |
| Square Matrix | Non-Square Matrix |
| matrix1 = Matrix(2, 2)  matrix1.data = [[1, 2], [4, 5]]  matrix1.removeCol(1)  matrix1.data == [[1], [4]] | matrix1 = Matrix(2, 3)  matrix1.data = [[1, 2, 3], [4, 5, 6]]  matrix1.removeCol(2)  matrix1.data == [[1, 2], [4, 5]] |

| **Constructor (init)** |  |
| --- | --- |
| Square Matrix | Non-Square Matrix |
| matrix1 = Matrix(2, 2)  matrix1.data = [[1, 2], [4, 5]] | matrix1 = Matrix(2, 3)  matrix1.data = [[1, 2, 3], [4, 5, 6]] |

## Problem 3 - Expression Combination - 25 Points

Implement the recursive function *expression(numbers)* to generate all possible valid expressions that can be formed using a list of integers and operators. The operators are addition (+), subtraction (-), and multiplication (\*), and they can be used between any two adjacent integers. The function should return a list of all possible valid expressions that can be formed using the given integers and operators.

**Example 1**

res = expression([1,2,3])

print(res ) # Should print: ["1+2+3", "1+2-3", "1+2\*3", "1-2+3",

# "1-2-3", "1-2\*3", "1\*2+3", "1\*2-3", "1\*2\*3"]

**Example 2**

res = expression([8,9])

print(res ) # Should print: ["8+9", "8\*9", "8-9"]

**Requirements**

* You have to use recursion.
* The space complexity of your implementation can be at most **O(3^(n-1))**.
* You can return the list elements in any order.

## Problem 4 - Maze Path - 25 Points

Given a maze represented by a 2D array of integers, write the recursive function solver(maze) to find a path from the starting point (top-left corner) to the ending point (bottom-right corner). The maze contains obstacles represented by 0s and free spaces represented by 1s. The function should return a list of tuples representing the path coordinates from start to end or an empty list if no path exists. We define the priority of each direction: down > right > up > left. Your solution should choose the direction with the highest priority if multiple directions are available.

**Example 1**

res = solver([[1, 0, 1, 1, 1],

[1, 0, 1, 0, 1],

[1, 0, 1, 0, 1],

[1, 1, 1, 0, 1]])

print(res) # Should print: [(0, 0), (1, 0), (2, 0), (3, 0),

# (3, 1), (3, 2), (2, 2),(1, 2), (0, 2), (0, 3),

# (0, 4), (1, 4), (2, 4), (3, 4)]

**Example 2**

res = solver([[1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],

[0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1],

[1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1],

[1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1],

[1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1],

[1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1],

[1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1],

[1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1],

[1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1],

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])

print(res ) # Should print: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2),

# (3, 2), (4, 2), (5, 2), (5, 3), (5, 4), (4, 4), (3, 4),

# (2, 4), (1, 4), (0, 4), (0, 5), (0, 6), (0, 7), (1, 7),

# (2, 7), (3, 7), (4, 7), (4, 8), (4, 9), (4, 10), (3, 10),

# (2, 10), (2, 11), (2, 12), (3, 12), (4, 12), (5, 12),

# (6,12), (7, 12), (8, 12), (9, 12), (9, 13), (9, 14)]

**Requirements**

* You have to use recursion.
* The space complexity of your implementation can be at most **O(n^2)** (auxiliary).
* The time complexity of your implementation can be at most **O(n^2)**.