Assignment 1 (8%)

Due: Thursday, November 28, 2019, 5:00 pm - Week 6

Objectives

The objectives of this assignment are:

- To gain experience in designing algorithms for a given problem description and implementing those algorithms in Python.
- To demonstrate your understanding on:
  - How to decompose code into functions in Python.
  - How to read from text files using Python.
  - How to manipulate lists using basic operations.

Submission Procedure

Your assignment will not be accepted unless it meets these requirements:

1. Your name and student ID must be included at the start of every file in your submission.
2. Save your files into a zip file called YourStudentID.zip
3. Submit your zip file containing your entire submission to Moodle.

Late Submission

Late submission will have 5% deduction of the total assignment marks per day (including weekends). Assignments submitted 7 days after the due date will not be accepted.

Important Notes

- Please ensure that you have read and understood the university's procedure on plagiarism and collusion available at https://www.monashcollege.edu.au/__data/assets/pdf_file/0005/1266845/Student-Academic-Integrity-Diplomas-Procedure.pdf . You will be required to agree to these policies when you submit your assignment.
- Your code will be checked against other students' work and code available online by advanced plagiarism detection systems. Make sure your work is your own. Do not take the risk.
- Your program will be checked against a number of test cases. Do not forget to include comments in your code explaining your algorithm. If your implementation has bugs, you may still get some marks based on how close your algorithm is to the correct algorithm. This is made difficult if code is poorly documented.
- Where it would simplify the problem you may not use built-in Python functions or libraries (e.g. using list.sort() or sorted()). Remember that this is an assignment focusing on algorithms and programming.

Assignment code interview

Each student will be interviewed during a lab session regarding their submission to gauge your personal understanding of your Assignment code. The purpose of this is to ensure that you have completed the code yourself and that you understand the code submitted. Your assignment mark will be scaled according to the responses provided.

Interview Rubric

0	The student cannot answer even the simplest of questions. There is no sign of preparation. They probably have not seen the code before.
0.25	There is some evidence the student has seen the code. The answer to a least one question contained some correct points. However, it is clear they cannot engage in a knowledgeable discussion about the code.
0.5	The student seems underprepared. Answers are long-winded and only partly correct. They seem to be trying to work out the code as they read it. They seem to be trying to remember something they were told but now can't remember. However, they clearly know something about the code. With prompting, they fail to improve on a partial or incorrect answer.
0.75	The student seems reasonably well prepared. Questions are answered correctly for the most part but the speed and/or confidence they are answered with is not 100%. With prompting, they can add a partially correct answer or correct an incorrect answer.
1	The student is fully prepared. All questions were answered quickly and confidently. It is absolutely clear that the student has performed all of the coding themselves.

Marking Criteria

This assignment contributes to 8% of your final mark.

- Part A: Representation and display (15 marks)
- Part B: Helper functions (25 marks)
- Part C: Generating random board (30 marks)
- Part D: Decomposition, Variable names and code Documentation (10 marks)

Total: 80 marks

Magnets Puzzle

The puzzle game Magnets involves placing a set of magnet blocks in pre-arranged empty slots on a board to satisfy a set of constraints.

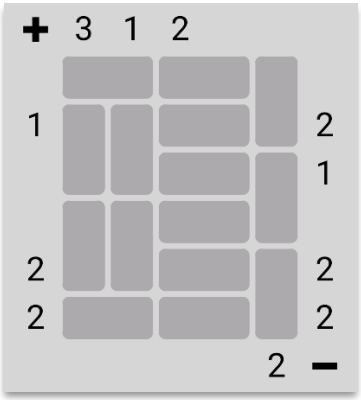


Figure 1 - Example: Unsolved Board with empty slots

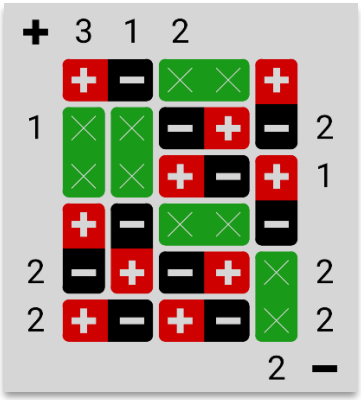


Figure 2 - Example: Solved board all slots are filled with magnet blocks.

Each slot can contain either a non-magnetic blank block (indicated by two 'X's), or a magnet block with a positive (+) pole and a negative (-) pole. The numbers along the left and top sides of the board show the number of required positive poles (+) in a particular row or column. Similarly, the numbers along the right and bottom show the number of negative poles (-) that are necessary in a particular row or column. (See Fig. 2)

Rows and columns without a number at one or both ends can have any number of positive (+) or negative (-) poles.

In addition, a puzzle solution must satisfy the constraint that horizontally and vertically adjacent squares in neighbouring blocks cannot have the same poles (See Fig. 3). Diagonally joined squares are not limited this way as shown in Fig. 4.

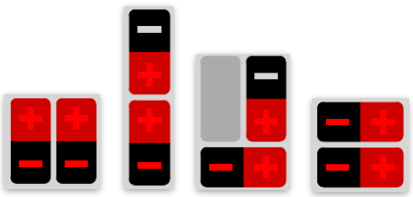


Figure 3 - Examples of Invalid Magnet Block Placements

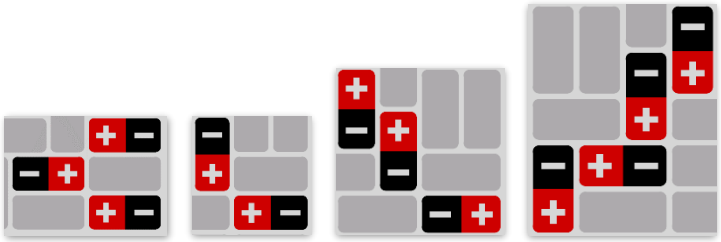


Figure 4 - Examples of Valid Magnet Block Placements

Part A: Representation and display (15 marks)

Magnets will require the following data structures:

positivesColumn, **negativesColumn**, **positivesRow** and **negativesRow** lists, which indicate the number of relevant poles. The positive (+) poles must be shown along the top and left edges and the negative (–) poles must be shown on the bottom and right edges. Values of -1 indicate that the corresponding row or column is unconstrained and can have any number of positive (+) or negative (–) poles.

Slots represent an empty space that can accommodate magnetic or blank blocks. Squares represent a single square on the board i.e. half a slot or half a block. All blocks and slots consist of two squares.

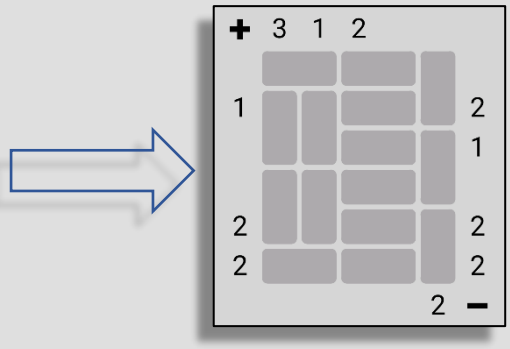
orientations, represents a list of lists with all pre-arranged slots on an board. Orientations' lists represent a single row on the board and it will contain one of the following characters, 'T', 'B', 'L', 'R'.

- A vertical slot will contain the letters 'T' and 'B' that represent the top and bottom of the slot.
- A horizontal slot will contain the letters 'L' and 'R' that represent the left and right sides of the slot.

M and **N** represent the size of the board i.e. **M** rows * **N** columns.

See the example below:

```
M   = 6 # number of rows
N   = 5 # number of columns
positivesColumn = [3, 1, 2, -1, -1]
negativesColumn = [-1, -1, -1, -1, 2]
positivesRow    = [-1, 1, -1, -1, 2, 2]
negativesRow    = [-1, 2, 1, -1, 2, 2]
orientations    =[['L', 'R', 'L', 'R', 'T'],
                  ['T', 'T', 'L', 'R', 'B'],
                  ['B', 'B', 'L', 'R', 'T'],
                  ['T', 'T', 'L', 'R', 'B'],
                  ['B', 'B', 'L', 'R', 'T'],
                  ['L', 'R', 'L', 'R', 'B']]
```



Example 1 – Sample variables for input parameters

Note: These values are not restricted to the example given above and they could be any number. Your code should be able to handle any board configuration within the appropriate constraints.

For simplicity, you can restrict your testing to the following board sizes:

M	N
4	3
4	5
6	5
6	7

Task 1: Initial setup (5 marks)

Write the function **readBoard(fileName)** which takes as input the name of a file in which the board is stored and returns **positivesColumn**, **negativesColumn**, **positivesRow** , **negativesRow** , **orientations**, and **workingBoard** as shown in Example 1 in the previous section.

workingBoard represents a partial solution to the board as a list of lists (see Example 3 in Part A – Task 2), where each square contains one of the following characters '+', '-', 'X', and 'E'. 'E' represents the empty squares and 'X' represents blank (non-magnetic) blocks.

The following shows a sample file:

```
6
5
3 1 2 -1 -1
-1 -1 -1 -1 2
-1 1 -1 -1 2 2
-1 2 1 -1 2 2
LRLRT
TTLRB
BBLRT
TTLRB
BBLRT
LRLRB
+-XXE
XXEEE
XXEE+
+EXX-
-EEEX
+-+-X
```

Example 2 – Sample File

This file must have the following format:

- Lines 1 and 2: Represent **M** and **N** respectively
- Lines 3 to 6: Represent **positivesColumn**, **negativesColumn**, **positivesRow** and **negativesRow**, respectively.
- Lines 7 to (7+**M**-1): Represent the **orientations**
- **M** Remaining lines: Represent the **workingBoard**

Task 2: Display (10 marks)

Extend your program to print a board on the screen using the function **printBoard(positivesColumn**, **negativesColumn**, **positivesRow** , **negativesRow** , **orientations**, **workingBoard)**.

This function prints the content of the **workingBoard** to the screen except it will print a blank space for any square that contains 'E'. The numbers in **positivesColumn**, **negativesColumn**, **positivesRow** and **negativesRow** are also printed if they are not equal to -1.

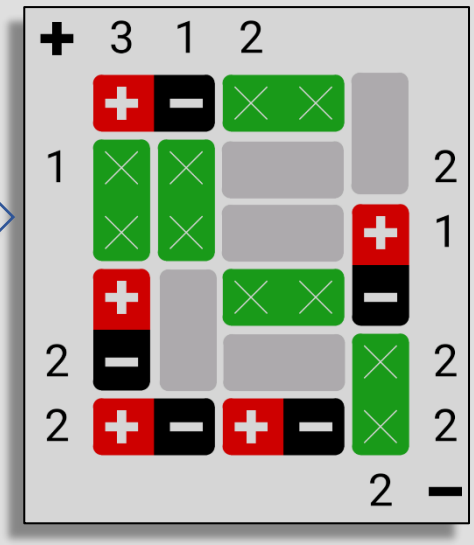
An example of **workingBoard** and the expected output of **printBoard** is given below. Your board visualisation must follow the formatting of the example below:

```
M  = 6 # number of rows
N  = 5 # number of columns
positivesColumn = [3, 1, 2, -1, -1]
negativesColumn = [-1, -1, -1, -1, 2]
positivesRow    = [-1, 1, -1, -1, 2, 2]
negativesRow    = [-1, 2, 1, -1, 2, 2]
orientations    =[['L', 'R', 'L', 'R', 'T'],
                  ['T', 'T', 'L', 'R', 'B'],
                  ['B', 'B', 'L', 'R', 'T'],
                  ['T', 'T', 'L', 'R', 'B'],
                  ['B', 'B', 'L', 'R', 'T'],
                  ['L', 'R', 'L', 'R', 'B']]
```

```
workingBoard    =[['+', '-', 'X', 'X', 'E'],
                  ['X', 'X', 'E', 'E', 'E'],
                  ['X', 'X', 'E', 'E', '+'],
                  ['+', 'E', 'X', 'X', '-'],
                  ['-', 'E', 'E', 'E', 'X'],
                  ['+', '-', '+', '-', 'X']]
printBoard(positivesColumn, negativesColumn, positivesRow , negativesRow ,
orientations, workingBoard)
```

```
 + | 3 | 1 | 2 |   |   |
---|---|---|---|---|---|---
   | +   - | X   X |   |
---|---|---|---|---|   |---
 1 | X | X |       |   |2
---|   |   |---|---|---|---
   | X | X |       | + |1
---|---|---|---|---|   |---
   | + |   | X   X | - |
---|   |   |---|---|---|---
 2 | - |   |       | X |2
---|---|---|---|---|   |---
 2 | +   - | +   - | X |2
---|---|---|---|---|---|---
   |   |   |   | 2 | - |
```

Similar to



Example 3- Sample output for printBoard function

Similarly, you can use the task 1's **readBoard** function to achieve the same result:

```
positivesColumn, negativesColumn, positivesRow , negativesRow , orientations,
workingBoard = readBoard("sampleFile1.txt")

printBoard(positivesColumn, negativesColumn, positivesRow , negativesRow ,
orientations, workingBoard)
```

```
 + | 3 | 1 | 2 |   |   |
---|---|---|---|---|---|---
   | +   - | X   X |   |
---|---|---|---|---|   |---
 1 | X | X |       |   |2
---|   |   |---|---|---|---
   | X | X |       | + |1
---|---|---|---|---|   |---
   | + |   | X   X | - |
---|   |   |---|---|---|---
 2 | - |   |       | X |2
---|---|---|---|---|   |---
 2 | +   - | +   - | X |2
---|---|---|---|---|---|---
   |   |   |   | 2 | - |
```

Example 4- Sample output for printBoard function using readBoard function first

Part B: Helper functions (25 marks)

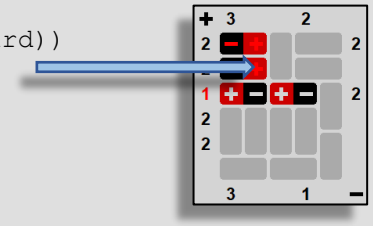
In this part, you will code several helper functions that you will need later.

Task 1: Safe placing (10 marks)

Write the function **canPlacePole(row, col, pole, workingBoard)** to check if it is safe to place one given magnetic block's pole for a particular coordinates. This function checks the surrounding squares and it will return a Boolean value. (See Figures 3 and 4 for more details.)

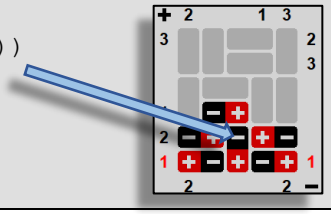
row and **col** represent the coordinates of a given pole. **pole** is a single character representation for the given polarity (i.e. '+' or '-') of one magnetic block's end. If the surrounding squares in **workingBoard** allow the placement of the given **pole** of a magnetic block then the function will return **True** otherwise it will return **False.** See the examples below:

```
>>> print(canPlacePole(1, 1, '+', workingBoard))
False
```



Example 4 - Example use of canPlacePole function for an existing workingBoard, positive (+) pole, row=1 and col=1

```
>>> print(canPlacePole(2, 3, '-', workingBoard))
True
```



Example 5- Example use of canPlacePole function for another existing workingBoard, negative (-) pole, row=2 and col=3

Hint: You can assume that square for the given coordinates is empty.

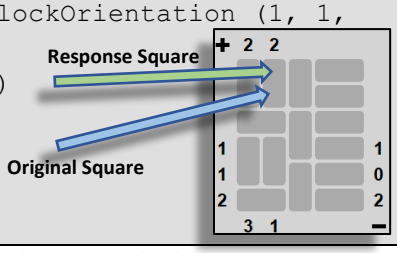
Task 2: Block orientation (5 marks)

Write the function **getBlockOrientation(row, col, orientations)** to get the orientation of a given slot as well as the coordinates of the other end of the slot, based on the given coordinates of a square and the **orientations**.

row and **col** represent the coordinates of a given square. **orientations,** represents a list of lists with all pre-arranged slots on the board. The function will return "TB" for top-bottom (vertical) blocks and "LR" for left-right (horizontal) blocks, it will also return the coordinates of the opposite end for that block. It must follow this exact order: resultOrientation, oppositeRow, and oppositeCol.

Note: The response for the slot's orientation can only be "TB" or "LR".

```
>>> resultOrientation, oppositeRow, oppositeCol = getBlockOrientation (1, 1,
orientations)
>>> print (resultOrientation, oppositeRow, oppositeCol)
'TB' 0 1
```



Example 6- Example use of getBlockOrientation function with an existing orientations board, row=1, and col=1

Task 3: Pole Count (5 marks)

Write the function **poleCount(rowOrCol, index, pole, workingBoard)** to find the total number of positive (+) poles or negative (-) poles in a particular row or column.

rowOrCol can only be 'r' or 'c' to indicate whether to check a given row or column, **index** indicates the index number for the row or column that function will check. Note: it must starts from zero.
pole can only be '+' or '-' to indicate which magnetic pole the function will be looking for; and consistent as before **workingBoard** represents a partial solution to the board as a list of lists.

Example behaviour of this function is demonstrated below:

```
 +  | 3 | 1 | 2 |   |   |
----|---|---|---|---|---|---
    | +   - | X   X |   |
----|---|---|---|---|   |---
 1  | X | X |       |   |2
----|   |   |---|---|---|---
    | X | X |       | + |1
----|---|---|---|---|   |---
    | + |   | X   X | - |
----|   |   |---|---|---|---
 2  | - |   |       | X |2
----|---|---|---|---|   |---
 2  | +   - | +   - | X |2
----|---|---|---|---|---|---
    |   |   |   |   | 2 | -

>>> workingBoard  = [ ['+', '-', 'X', 'X', 'E'],
                      ['X', 'X', 'E', 'E', 'E'],
                      ['X', 'X', 'E', 'E', '+'],
                      ['+', 'E', 'X', 'X', '-'],
                      ['-', 'E', 'E', 'E', 'X'],
                      ['+', '-', '+', '-', 'X'] ]

>>> print(poleCount ('r',5,'+', workingBoard))
2
>>> print(poleCount ('r',4,'-', workingBoard))
1
>>> print(poleCount ('c',4,'-', workingBoard))
1
```

Example 7 - examples of poleCount function

Task 4: Random Magnetic Pole Distribution (5 marks)

Write the function **randomPoleFlip(alist, percentage, flipValue)** to randomly swaps a percentage of elements from **alist** (which can represent one of **positivesColumn**, **negativesColumn**, **positivesRow,** or **negativesRow)** with the **flipValue.**

The number of elements that needs to be flipped is defined by the floor of **percentage** times the length of the **aList**.

```
alist=[1,2,3,4,5,6,7,8,9,10]
randomPoleFlip(alist,0.5,-1)
print(alist)
[1, -1, -1, 4, 5, 6, -1, -1, 9, -1]
```

Example 8 - In the example above, 50% of the elements are flipped to -1.

```
alist=[1,2,3,4,5,6,7,8,9,10]
randomPoleFlip(alist,0.35,-1)
print(alist)
[-1, 2, -1, 4, 5, 6, 7, 8, 9, -1]
```

*Example 9 - In the example above, 3 elements are flipped because the function uses floor of 0.35 * len(alist).*

Part C: Board Generation Functions (30 marks)

Task 1: Orientations generation (10 marks)

In this part you will write the function **orientationsGenerator(M,N)** to randomly generate the orientations as a list of lists for a board of size **M** x **N**.

The procedure of generating the orientations is as follows:

(Please note: **M** must be an even number, otherwise this procedure cannot be done).

1. The function starts with generating an M by N board with all orientations set as vertical blocks. See example below:

```
[['T', 'T', 'T', 'T', 'T'],
 ['B', 'B', 'B', 'B', 'B'],
 ['T', 'T', 'T', 'T', 'T'],
 ['B', 'B', 'B', 'B', 'B']]
```

Example 10 - step 1 orientations list when M=4 and N=5

2. The function continues by picking a random block:
   a. If it is a LR block, check the block that is immediately above it. If it is also an LR block and it is perfectly aligned with the picked block then change both blocks to become TB blocks. If it is not possible to do this with the top block, then try with the bottom block.
   b. If it is a TB block, check the block that is immediately to its left. If it is also a TB block and it is perfectly aligned with the picked block then change both blocks to become LR blocks. If it is not possible to do this with the left block, then try with the right block.
   c. If the blocks are not of the same type or not aligned then you will go to the next step.
3. Repeat the process at least 1000 times to make certain it is sufficiently random. See the figure below of how this procedure is done.

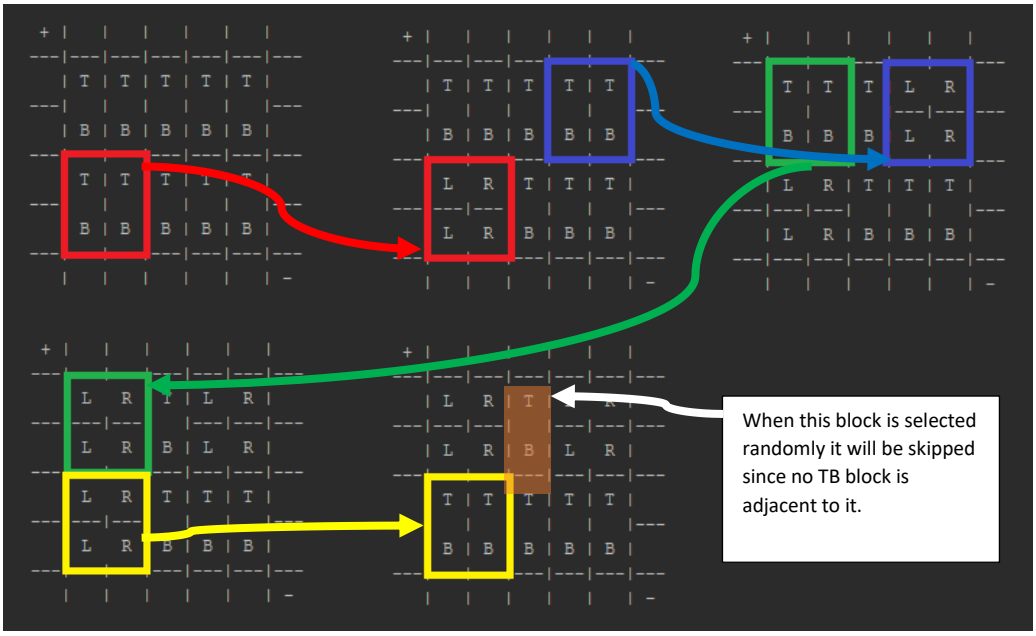


Figure 5 - Orientations Generation Sample Process

4. Return orientations as a list of lists

Example behaviour of this function is demonstrated below (note the result are random):

```
M=4
N=5
print(orientationsGenerator(M, N))

[['T', 'L', 'R', 'T', 'T'],
 ['B', 'T', 'T', 'B', 'B'],
 ['T', 'B', 'B', 'T', 'T'],
 ['B', 'L', 'R', 'B', 'B']]


M=6
N=5
print(orientationsGenerator(M, N))

[['L', 'R', 'T', 'L', 'R'],
 ['L', 'R', 'B', 'L', 'R'],
 ['L', 'R', 'L', 'R', 'T'],
 ['L', 'R', 'T', 'T', 'B'],
 ['T', 'T', 'B', 'B', 'T'],
 ['B', 'B', 'L', 'R', 'B']]
```

Example 11 - Two example outputs of orientationsGenerator functions

Hint: You can use the **printBoard** function from Part A from a better display of the orientations.

Task 2: Filling board with magnets (10 marks)

Write the function **fillWithMagnets(orientations)**, that creates a board that is full of magnet blocks.

You need to use the helper functions from PART B and the process must follow the following steps:

1. Create an empty workingBoard.
2. It will then scan each row of **orientations** from top left corner row by row to the bottom right corner to fill the corresponding slots in the board with magnet blocks. Check if there is an existing pole present in the square that is being checked. If there is a block present, skip to the next square. Otherwise:

   2.1. For an LR block:

   I. You need to check if it is safe to place a positive (+) pole in the left ('L') square of the block. If safe, place it and then negative (–) pole in the right ('R') square of the same block.

   II. Otherwise place the negative (-) pole in the left square and the positive (+) pole in the right square of the same block.

   2.2. For a TB block:

   I. You need to check if it is safe to place a positive (+) pole in the top ('T') part of the block. If safe, place the positive (+) pole in top square and the negative (–) pole in bottom ('B') square of the same block.

   II. Otherwise place the negative (-) pole in top square and the positive (+) pole in the bottom square of the same block.

   Note: Here you must check individual squares (i.e. single end of a block). You need to use your **canPlacePole** and **getBlockOrientation** functions from PART B.

3. Return the resulting workingBoard

The behaviour of this function is as following:

```
positivesColumn = [-1, -1, -1, -1, -1]
negativesColumn = [-1, -1, -1, -1, -1]
positivesRow    = [-1, -1, -1, -1, -1]
negativesRow    = [-1, -1, -1, -1, -1]
orientations    = [ ['L', 'R', 'L', 'R', 'T'],
                    ['T', 'T', 'L', 'R', 'B'],
                    ['B', 'B', 'L', 'R', 'T'],
                    ['T', 'T', 'L', 'R', 'B'],
                    ['B', 'B', 'L', 'R', 'T'],
                    ['L', 'R', 'L', 'R', 'B'] ]

workingBoard= fillWithMagnets(orientations)

printBoard(positivesColumn, negativesColumn, positivesRow , negativesRow ,
orientations, workingBoard)

 + |   |   |   |   |   |
---|---|---|---|---|---|---
   | +   - | +   - | + |
---|---|---|---|---|   |---
   | - | + | -   + | - |
---|   |   |---|---|---|---
   | + | - | +   - | + |
---|---|---|---|---|   |---
   | - | + | -   + | - |
---|   |   |---|---|---|---
   | + | - | +   - | + |
---|---|---|---|---|   |---
   | -   + | -   + | - |
---|---|---|---|---|---|---
   |   |   |   |   |   | -
```

Example 12 – Example use of fillWithMagnets function

Task 3: Generating random new board (10 marks)

Write the function **randomNewBoard (M,N)**. This function uses the previous functions to create a new random boardand and returns **positivesColumn**, **negativesColumn**, **positivesRow, negativesRow** , **workingBoard**, and **orientations**.

The procedure of generating a random board must follow these steps:

1. Generate the orientations board of size M x N randomly.
   Note: use **orientationsGenerator** function for this.
2. Fill the board with magnet blocks.
   Note: use **fillWithMagnets** function for this.
3. Randomly replace 30% of the blocks with blank blocks ('X' blocks)
4. Generate the resulting **positivesColumn**, **negativesColumn**, **positivesRow,** and **negativesRow** lists.
   Note: use **poleCount** function for this.
5. Replace 50% of the numbers in each of **positivesColumn**, **negativesColumn**, **positivesRow,** and **negativesRow** lists with -1.
   Note: use **randomPoleFlip** function for this.

Example behaviour of this function is as below:

```
M = 6
N = 5

positivesColumn, negativesColumn, positivesRow, and negativesRow, workingBoard,
orientations = randomNewBoard (M,N)

printBoard(positivesColumn, negativesColumn, positivesRow , negativesRow ,
orientations, workingBoard)

 +  | 1 |   |   | 3 | 2 |
----|---|---|---|---|---|---
    | X   X | + | -   + |
----|---|---|   |---|---|---
 2  | -  | +  | -  | +   - |
----|    |   |---|---|---|---
    | +  | -  | +  | -   + |2
----|---|---|   |---|---|---
    | X   X | -  | +   - |
----|---|---|---|---|---|---
 1  | X  | -  | +  | X   X |1
----|    |   |   |---|---|---
 2  | X  | +  | -  | +   - |2
----|---|---|---|---|---|---
    |   |   | 3 | 2 | 3 | -
```

*Example 13 – Example use of randomNewBoard function (**note**: The generated board will change with every run)*

Part D: Decomposition, Variable names and code Documentation (10 marks)

Marks will be allocated for good use of variable names, code documentation and proper decomposition.