MONASH College MCD4710 - Introduction to Algorithms and Programming

Assignment 2 (10%)

Due: Thursday, Jan 9, 2020, 5:00 pm - Week 10

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 towards 10% of your final mark.

- Part A: Helper functions (10 marks)
- Part B: Brute-force (30 marks) or: Backtracking (30 marks)
- Part C: Analysis and Discussion of Approach (10 marks)
- Part D: Decomposition, Variable names and Code Documentation (10 marks)

Total: 60 marks

In this assignment, we will continue with the Magnets puzzle from assignment 1.
***Note**: You will need the functions from assignment 1 to be able to complete this assignment.*
For consistency, use the provided base code for assignment 1 on Moodle.

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.

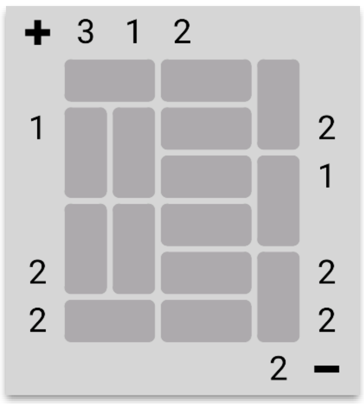

Figure 1 - Example: Unsolved Board with empty slots

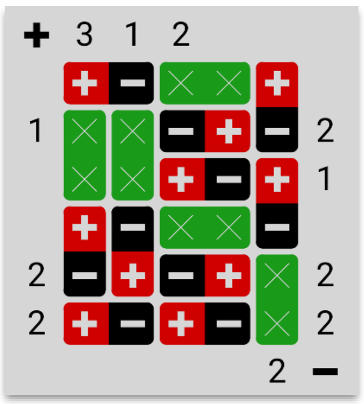

Figure 2 - Example: Solved board all slots are filled with magnet blocks.

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.

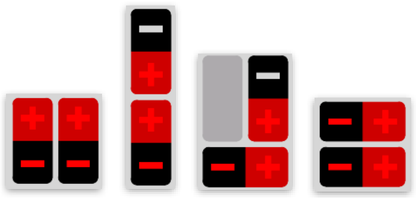

Figure 3 - Examples of Invalid Magnet Block Placements

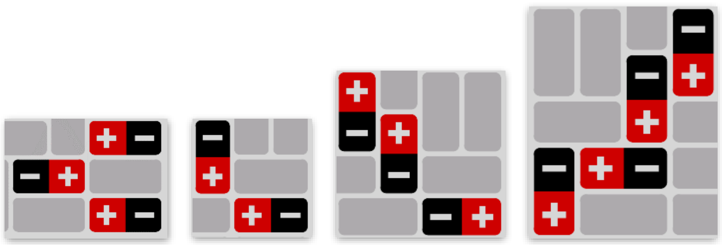

Figure 4 - Examples of Valid Magnet Block Placements

Part A: Helper functions (10 marks)

This task involves writing some new helper functions, that are useful to use with brute force and backtracking algorithms.

Task 1: Converting a set into a board (5 Marks)

Write a function **setToBoard(set,orientations)** that takes a given **set** list and converts it into a board based on existing **orientations**. This function will return the resulting board as the output.

set is a list of values that can only be 0, 1, or 2. The length of **set** is equivalent to the number of blocks that will be converted.

- 0 represents:
  - **For an LR block**: A positive pole ('+') in the left ('L') square and therefore negative pole ('-') in the right ('R') square.
  - **For a TB block**: A positive pole ('+') in the top ('T') square and therefore negative pole ('-') in the bottom ('B') square.
- 1 represents:
  - **For an LR block**: A negative pole ('-') in the left ('L') square and therefore positive pole ('+') in the right ('R') square.
  - **For a TB block**: A negative pole ('-') in the top ('T') square and therefore positive pole ('+') in the bottom ('B') square.
- 2 represents:
  - Blank block ('X') squares for both squares of any block orientation.

Important Notes:

- The function will start from the top left square find the required number of blocks that it will need to change based on the length of **set**.
- The **len (set) <= (M*N)//2**, where **(M*N)//2** stands for the number of slots in the entire board.
- If the **len(set)** is less than **(M*N)//2** then the board should turn the remaining slots to empty block('E')

Example of this function's use is as follows:

```
positivesColumn = [-1, -1, -1, -1, -1]
negativesColumn = [-1, -1, -1, -1, -1]
positivesRow    = [-1, -1, -1, -1, -1, -1]
negativesRow    = [-1, -1, -1, -1, -1, -1]
orientations    = [ ['T', 'L', 'R', 'L', 'R'],
                    ['B', 'T', 'T', 'L', 'R'],
                    ['T', 'B', 'B', 'T', 'T'],
                    ['B', 'L', 'R', 'B', 'B'],
                    ['L', 'R', 'L', 'R', 'T'],
                    ['L', 'R', 'L', 'R', 'B'] ]
printBoard(positivesColumn, negativesColumn, positivesRow, negativesRow, orientations,
orientations)
set=[0,1,1,0]
workingBoard= setToBoard(set,orientations)
print("###########################################")
printBoard(positivesColumn, negativesColumn, positivesRow, negativesRow, orientations,
workingBoard)
set =[0,2,0,2,1,0,2]
workingBoard= setToBoard(set,orientations)
print("###########################################")
printBoard(positivesColumn, negativesColumn, positivesRow, negativesRow, orientations,
workingBoard)
```

```
 + |   |   |   |   |   |
---|---|---|---|---|---|---
   | T | L   R | L   R |
---|   |---|---|---|---|---
   | B | T | T | L   R |
---|---|   |   |---|---|---
   | T | B | B | T | T |
---|   |---|---|   |   |---
   | B | L   R | B | B |
---|---|---|---|---|---|---
   | L   R | L   R | T |
---|---|---|---|---|   |---
   | L   R | L   R | B |
---|---|---|---|---|---|---
   |   |   |   |   |   | -
###########################################
 + |   |   |   |   |   |
---|---|---|---|---|---|---
   | + | -   + | -   + |
---|   |---|---|---|---|---
   | - | + |   |       |
---|---|   |   |---|---|---
   |   | - |   |   |   |
---|   |---|---|   |   |---
   |   |       |   |   |
---|---|---|---|---|---|---
   |       |       |   |
---|---|---|---|---|   |---
   |       |       |   |
---|---|---|---|---|---|---
   |   |   |   |   |   | -
###########################################
 + |   |   |   |   |   |
---|---|---|---|---|---|---
   | + | X   X | +   - |
---|   |---|---|---|---|---
   | - | X | - | +   - |
---|---|   |   |---|---|---
   | X | X | + |   |   |
---|   |---|---|   |   |---
   | X |       |   |   |
---|---|---|---|---|---|---
   |       |       |   |
---|---|---|---|---|   |---
   |       |       |   |
---|---|---|---|---|---|---
   |   |   |   |   |   | -
```

Example 1 – setToBoard function example

Task 2: Check solution (5 marks)

Write the function **isSolution(positivesColumn, negativesColumn, positivesRow, negativesRow, orientations, workingBoard)** that checks if a given board is a valid solution to the puzzle. The function will return Boolean value True for valid solutions; otherwise, it will return False.

The function should check the following:

- The number of positive poles (+) and negative poles (–)in each row and column of the board must match **positivesColumn, negativesColumn, positivesRow,** and **negativesRow**
- The given board does not violate the orthogonal rule i.e. the magnetic poles are not vertically or horizontally adjacent.

Example behaviour of this function is demonstrate below:

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

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

workingBoard    = [['+', '-', '+'],
                    ['-', '+', '-']]

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

 + | 1 | 1 | 1 |
---|---|---|---|---
   | +   - | - |
---|---|---|   |---
   | -   + | + |
---|---|---|---|---
   | 1 | 1 | 1 | -
False (because it violates the orthogonal rule, see column three)
 + | 1 | 1 | 1 |
---|---|---|---|---
   | +   - | + |
---|---|---|   |---
   | -   + | - |
---|---|---|---|---
   | 1 | 1 | 1 | -
True
```

Example 2 - example use of isSolution function

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

```
workingBoard     = [['+', '-', '+'],
                    ['-', '+', '-']]

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

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

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

 + | 1 | 1 |   |
---|---|---|---|---
 1 | +   - | + |
---|---|---|   |---
   | -   + | - |
---|---|---|---|---
   | 1 | 1 |   | -
False (because the first row has 2 '+' instead of 1)
 + | 1 | 1 |   |
---|---|---|---|---
 1 | +   - | X |
---|---|---|   |---
   | -   + | X |
---|---|---|---|---
   | 1 | 1 |   | -
True
```

Example 3 - example use of isSolution function

Part B:

Choose either task 1 or task 2 in part B.

In this part, you can **choose one** of the following options:

1. Option 1: Task 1: Brute-force (30 marks) **or**
2. Option 2: Task 2: Backtracking (30 marks)

Note: Only attempt one; you are not required to do both.

Task 1: Brute-force (30 marks)

This task is about applying the brute-force approach to find solutions for the magnet puzzle.

Write the function **bruteforce(positivesColumn, negativesColumn, positivesRow, negativesRow, orientations)** which will return the solution to the board once it finds it. The function will find the solution using the bruteforce approach by exploring all possible solutions. You will need to use the **set** values of 0, 1, or 2 as described in task 1, part A (page 4).

Hint: In order to generate the solutions you need to list them in a Lexicographical order (https://en.wikipedia.org/wiki/Lexicographical_order).

For example, for a board of size 2x3 we have 3 blocks, and all possible solutions are:

000	100	200
001	101	201
002	102	202
010	110	210
011	111	211
012	112	212
020	120	220
021	121	221
022	122	222

Table 1 - Sample bruteforce in-order generated subsets

Example behaviour of this function is demonstrated below:

```
positivesColumn = [2, 1, 1, -1, -1]
negativesColumn = [1, -1, -1, 2, 1]
positivesRow    = [2, -1, -1, 1]
negativesRow    = [1, 3, -1, -1]

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

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

```
print("#############################################")

solution=[]

solution = bruteforce(positivesColumn, negativesColumn, positivesRow, negativesRow, 
orientations)

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




 + | 2 | 1 | 1 |   |   |
---|---|---|---|---|---|---
 2 | T | L   R | L   R |1
---|   |---|---|---|---|---
   | B | L   R | L   R |3
---|---|---|---|---|---|---
   | L   R | L   R | T |
---|---|---|---|---|   |---
 1 | L   R | L   R | B |
---|---|---|---|---|---|---
   | 1 |   |   | 2 | 1 | -
#############################################
 + | 2 | 1 | 1 |   |   |
---|---|---|---|---|---|---
 2 | + | X   X | -   + |1
---|   |---|---|---|---|---
   | - | +   - | +   - |3
---|---|---|---|---|---|---
   | +   - | +   - | X |
---|---|---|---|---|   |---
 1 | X   X | -   + | X |
---|---|---|---|---|---|---
   | 1 |   |   | 2 | 1 | -
```

Example 4 - Example use of bruteforce function

Task 2: Backtracking (30 marks)

Write the function **backtrack(positivesColumn, negativesColumn, positivesRow, negativesRow, orientations, partial)** that uses the backtracking approach to find a solution to the Magents puzzle. The function will return the solution as soon as it finds it.

Example behaviour of this function is demonstrated below:

```
positivesColumn = [2, 1, 1, -1, -1]
negativesColumn = [1, -1, -1, 2, 1]
positivesRow    = [2, -1, -1, 1]
negativesRow    = [1, 3, -1, -1]



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

printBoard(positivesColumn, negativesColumn, positivesRow, negativesRow,
orientations, orientations)
print("#############################################")

partial=[]
solution = []
solution = backtrack(positivesColumn, negativesColumn, positivesRow, negativesRow,
orientations, partial)


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

 + | 2 | 1 | 1 |   |   |
---|---|---|---|---|---|---
 2 | T | L   R | L   R |1
---|   |---|---|---|---|---
   | B | L   R | L   R |3
---|---|---|---|---|---|---
   | L   R | L   R | T |
---|---|---|---|---|   |---
 1 | L   R | L   R | B |
---|---|---|---|---|---|---
   | 1 |   |   | 2 | 1 | -
#############################################
 + | 2 | 1 | 1 |   |   |
---|---|---|---|---|---|---
 2 | + | X   X | -   + |1
---|   |---|---|---|---|---
   | - | +   - | +   - |3
---|---|---|---|---|---|---
   | +   - | +   - | X |
---|---|---|---|---|   |---
 1 | X   X | -   + | X |
---|---|---|---|---|---|---
   | 1 |   |   | 2 | 1 | -
```

Part C: Approach and Analysis (10 marks)

This part must be delivered a separate Word or PDF document, using the following format:

LastNameID_PartD.docx or **pdf**. Make sure that it is included in same zip folder as your code.

Task 1: Description and justification of approach

Describe the brute-force or backtracking approaches that you implemented in part B, task 1 (on page 9) or part B, task 2 (on page 11). Your discussion must directly relate to each of these algorithms as you have used them to solve the Magnets puzzle. Make sure that you address the following:

1. How the individual possible or partial (if applicable) solutions are generated and represented
2. What are the complexities of your algorithm (include descriptions and the Big O notation)
3. Discuss whether you can be certain that the algorithm will give the correct solution. In your discussion, include why you can or cannot be certain.

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

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