Assignment 2 (10%)

Due: Thursday, May 7, 2019, 11:55 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 2 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

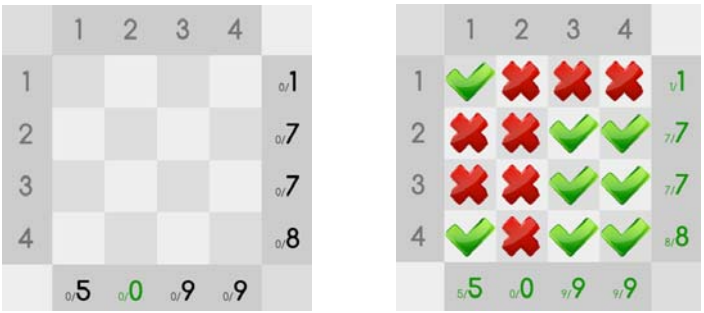
Total: 60 marks

- **A.** Task 1: Helper functions (15 marks)
- **B.** Task 2: Brute-force (25 marks) or Backtracking (25 marks)
- **C.** Task 3: Description and justification of approach (10 marks)
- **D.** Task 4: Decomposition, Variable names and Code Documentation (10 marks)

Background: Kakurasu Puzzle

Write all the code for this task in one file and name it kakurasu.py

In this assignment, you are required to write brute force or backtracking algorithms to solve Kakurasu puzzle. This puzzle is played on a square grid. The goal is to tick some cells to satisfy the clues around the grid. For example, the puzzle on the left has the solution shown on the right:



(a) (b)

Fig. 1: Kakurasu puzzle, (a) unsolved (b) solved.

The numbers at the top and at the left are the values for each of the squares in the rows and columns (example: the first square in a row or column is worth 1, the second 2, the third 3, etc.).

The numbers at the bottom and at the right equal the row and column totals for the ticked squares. Ticking a square will add that square's value to both the row's total and the column's total.

Task 1: Helper functions (15 marks)

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

Task 1A: printing board (5 marks)

Write a function `printBoard` to print the board on screen. This function prints the content of the board to the screen along with the headers. An example of board and the expected output of `printBoard` is given below. Your board visualisation must follow the formatting of the example below.

Function header: `printBoard(board, rowSum, colSum)`

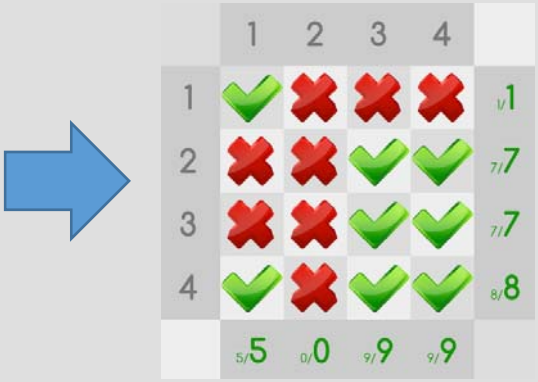
Input: board as list of lists of size n * n, where 1 corresponds to a ticked square and 0 corresponds to a crossed square; `rowSum` as a list contain the sum of each row; `colSum` as a list contain the sum of each column.

Output: print the board on screen with correct top, bottom, left, and right headers.

Examples:

```
rowSum = [1, 7, 7, 8]
colSum = [5, 0, 9, 9]
board = [[1, 0, 0, 0], [0, 0, 1, 1], [0, 0, 1, 1], [1, 0, 1, 1]]
printBoard(board, rowSum, colSum)
```

```
   | 1 | 2 | 3 | 4 |
---|---|---|---|---|---
 1 | 1 | 0 | 0 | 0 | 1
---|---|---|---|---|---
 2 | 0 | 0 | 1 | 1 | 7
---|---|---|---|---|---
 3 | 0 | 0 | 1 | 1 | 7
---|---|---|---|---|---
 4 | 1 | 0 | 1 | 1 | 8
---|---|---|---|---|---
   | 5 | 0 | 9 | 9 |
```



Task 1B: converting a bit list into a board (5 marks)

Write a function `list2board(L)` that takes a bit list and convert it to a board. This function will return a board as output. The bit list L is a list of elements that are either 0,1:

0 means: crossed square. 

1 means: ticked square. 

Function header: `list2board(L)`

Input: L as a list of size n^2, that contains ones and zeros as explained above.

Output: board as a list of lists of size n * n. That contains ones and zeros as explained above.

Examples:

```
Example 1:

L = [1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1]
board = list2board(L)
rowSum = [1, 7, 7, 8]
colSum = [5, 0, 9, 9]
print(board)
printBoard(board, rowSum, colSum)

[[1, 0, 0, 0], [0, 0, 1, 1], [0, 0, 1, 1], [1, 0, 1, 1]]
   | 1 | 2 | 3 | 4 |
---|---|---|---|---|---
 1 | 1 | 0 | 0 | 0 | 1
---|---|---|---|---|---
 2 | 0 | 0 | 1 | 1 | 7
---|---|---|---|---|---
 3 | 0 | 0 | 1 | 1 | 7
---|---|---|---|---|---
 4 | 1 | 0 | 1 | 1 | 8
---|---|---|---|---|---
   | 5 | 0 | 9 | 9 |

Example 2:

L = [1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1]
board = list2board(L)
rowSum = [6,0,6,14,12]
colSum = [1,7,9,12,10]
print(board)
printBoard(board, rowSum, colSum)

[[1, 0, 0, 0, 1], [0, 0, 0, 0, 0], [0, 1, 0, 1, 0], [0, 1, 1, 1, 1], [0, 0, 1, 1, 1]]
   | 1 | 2 | 3 | 4 | 5 |
---|---|---|---|---|---|---
 1 | 1 | 0 | 0 | 0 | 1 | 6
---|---|---|---|---|---|---
 2 | 0 | 0 | 0 | 0 | 0 | 0
---|---|---|---|---|---|---
 3 | 0 | 1 | 0 | 1 | 0 | 6
---|---|---|---|---|---|---
 4 | 0 | 1 | 1 | 1 | 1 | 14
---|---|---|---|---|---|---
 5 | 0 | 0 | 1 | 1 | 1 | 12
---|---|---|---|---|---|---
   | 1 | 7 | 9 | 12| 10|
```

Task 1C: Check solution (5 marks)

Write a function `isSolution` that checks if a given board is a solution to the puzzle. If the board is a solution it returns True otherwise it returns False. The function should check if the sum of each row and column satisfy their respective constraints at the right and the bottom of the board.

Function header: `isSolution(board, rowSum, colSum):`

Input: board as list of lists of size n * n, where 1 corresponding to a ticked square and 0 corresponds to crossed square; `rowSum` as a list contain the sum of each row; `colSum` as a list contain the sum of each column.

Output: A Boolean value, True if the board is a solution to the puzzle, False otherwise.

Examples:

```
Example 1:

board = [[1, 1, 0, 0], [0, 0, 1, 1], [0, 0, 1, 1], [1, 0, 1, 1]]
rowSum = [1, 7, 7, 8]
colSum = [5, 0, 9, 9]
print(isSolution(board,rowSum, colSum))
printBoard(board, rowSum, colSum)
False
First row:  1 + 2 = 3 ≠ 1 ✗        First column:  1 + 4 = 5 ✔
Second row: 3 + 4 = 7 ✔             Second column: 1 ≠ 0 ✗
Third row:  3 + 4 = 7 ✔             Third column:  2 + 3 + 4 = 9 ✔
Fourth row: 1 + 3 + 4 = 8 ✔         Fourth column: 2 + 3 + 4 = 9 ✔

   | 1 | 2 | 3 | 4 |
---|---|---|---|---|---
 1 | 1 | 1 | 0 | 0 | 1
---|---|---|---|---|---
 2 | 0 | 0 | 1 | 1 | 7
---|---|---|---|---|---
 3 | 0 | 0 | 1 | 1 | 7
---|---|---|---|---|---
 4 | 1 | 0 | 1 | 1 | 8
---|---|---|---|---|---
   | 5 | 0 | 9 | 9 |

Example 2:
board = [[1, 0, 0, 0, 1], [0, 0, 0, 0, 0], [0, 1, 0, 1, 0], [0, 1, 1, 1, 1], [0, 0, 1,
1, 1]]
rowSum = [6,0,6,14,12]
colSum = [1,7,9,12,10]
print(isSolution(board,rowSum, colSum))
printBoard(board, rowSum, colSum)
True
First row:  1 + 5 =6 ✔             First column:  1 = 1 ✔
Second row: 0 = 0 ✔                 Second column: 3 + 4 = 7 ✔
Third row:  2 + 4 = 6 ✔             Third column:  4 + 5 = 9 ✔
```

```
Fourth row: 2 + 3 + 4 + 5 = 14 ✔    Fourth column: 3 + 4 + 5 = 12 ✔
Fifth row:  3 + 4 + 5 = 12 ✔        Fifth column:  1 + 4 + 5 = 10 ✔

   | 1 | 2 | 3 | 4 | 5 |
---|---|---|---|---|---|---
 1 | 1 | 0 | 0 | 0 | 1 | 6
---|---|---|---|---|---|---
 2 | 0 | 0 | 0 | 0 | 0 | 0
---|---|---|---|---|---|---
 3 | 0 | 1 | 0 | 1 | 0 | 6
---|---|---|---|---|---|---
 4 | 0 | 1 | 1 | 1 | 1 | 14
---|---|---|---|---|---|---
 5 | 0 | 0 | 1 | 1 | 1 | 12
---|---|---|---|---|---|---
   | 1 | 7 | 9 | 12| 10|
```

Task 2:

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

1. Option 1: Task 2A: Brute-force (25 marks) or
2. Option 2: Task 2B: Backtracking (25 marks)

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

Task 2A: Brute-force (25 marks)

This task is about applying the brute-force approach to find a solution for the Kakurasu puzzle.

Prepare a python function `bruteforce`. The function will return the solved board once it finds it. The function will find the first correct solution using `bruteforce` approach by exploring all possible solutions. You will need to use bitlists that has bit values of 0, and 1. The definition of these bits is given in Task1, part B. For generating solutions, you need to list them in a Lexicographical order (https://en.wikipedia.org/wiki/Lexicographical_order). For example, for a board of size 2x2 all possible solutions are:

0, 0, 0, 0	0, 1, 0, 0	1, 0, 0, 0	1, 1, 0, 0
0, 0, 0, 1	0, 1, 0, 1	1, 0, 0, 1	1, 1, 0, 1
0, 0, 1, 0	0, 1, 1, 0	1, 0, 1, 0	1, 1, 1, 0
0, 0, 1, 1	0, 1, 1, 1	1, 0, 1, 1	1, 1, 1, 1

Function header: `bruteforce(rowSum, colSum)`

Input: `rowSum` as a list of size n contain the sum of each row; `colSum` as a list of size n contain the sum of each column.

Output: board as list of lists of size n * n, where 1 corresponding to a ticked square and 0 corresponds to crossed square.

Please note that your function should return the first solution as soon as you find it.

Examples:

```
# Example 1

rowSum = [1, 7, 7, 8]
colSum = [5, 0, 9, 9]
board = bruteforce(rowSum, colSum)
printBoard(board, rowSum, colSum)

   | 1 | 2 | 3 | 4 |
---|---|---|---|---|---
 1 | 1 | 0 | 0 | 0 | 1
---|---|---|---|---|---
 2 | 0 | 0 | 1 | 1 | 7
---|---|---|---|---|---
 3 | 0 | 0 | 1 | 1 | 7
---|---|---|---|---|---
 4 | 1 | 0 | 1 | 1 | 8
---|---|---|---|---|---
   | 5 | 0 | 9 | 9 |
```

```
# Example 2

rowSum = [8,9,6,9]
colSum = [4,9,10,7]
board = bruteforce(rowSum, colSum)
printBoard(board, rowSum, colSum)


   | 1 | 2 | 3 | 4 |
---|---|---|---|---|---
 1 | 1 | 0 | 1 | 1 | 8
---|---|---|---|---|---
 2 | 0 | 1 | 1 | 1 | 9
---|---|---|---|---|---
 3 | 1 | 1 | 1 | 0 | 6
---|---|---|---|---|---
 4 | 0 | 1 | 1 | 1 | 9
---|---|---|---|---|---
   | 4 | 9 | 10| 7 |

# Example 3

rowSum = [4,5,3,5]
colSum = [10,3,1,6]
board = bruteforce(rowSum, colSum)
printBoard(board, rowSum, colSum)


   | 1 | 2 | 3 | 4 |
---|---|---|---|---|---
 1 | 1 | 0 | 1 | 0 | 4
---|---|---|---|---|---
 2 | 1 | 0 | 0 | 1 | 5
---|---|---|---|---|---
 3 | 1 | 1 | 0 | 0 | 3
---|---|---|---|---|---
 4 | 1 | 0 | 0 | 1 | 5
---|---|---|---|---|---
   | 10| 3 | 1 | 6 |

# Example 4 (Warning this test case will take several minutes)

rowSum = [6,0,6,14,12]
colSum = [1,7,9,12,10]
board = bruteforce(rowSum, colSum)
printBoard(board, rowSum, colSum)
```

```
   | 1 | 2 | 3 | 4 | 5 |
---|---|---|---|---|---|---
 1 | 1 | 0 | 0 | 0 | 1 | 6
---|---|---|---|---|---|---
 2 | 0 | 0 | 0 | 0 | 0 | 0
---|---|---|---|---|---|---
 3 | 0 | 1 | 0 | 1 | 0 | 6
---|---|---|---|---|---|---
 4 | 0 | 1 | 1 | 1 | 1 | 14
---|---|---|---|---|---|---
 5 | 0 | 0 | 1 | 1 | 1 | 12
---|---|---|---|---|---|---
   | 1 | 7 | 9 | 12| 10|
```

Task 2B: Back tracking (25 marks)

Write a function backtrack, that uses the backtracking approach to find a solution to the Kakurasu puzzle. The function will return the **first solution as soon as it is found**.

Function header: `backtrack(rowSum, colSum)`

Input: `rowSum` as a list of size n contain the sum of each row; `colSum` as a list of size n contain the sum of each column.

Output: board as list of lists of size n * n, where 1 corresponding to a ticked square and 0 corresponds to crossed square.

Examples:

```
# Example 1

rowSum = [1,4,6,9]
colSum = [1,7,4,9]
board=backtrack(rowSum, colSum)
printBoard(board, rowSum, colSum)

   | 1 | 2 | 3 | 4 |
---|---|---|---|---|---
 1 | 1 | 0 | 0 | 0 | 1
---|---|---|---|---|---
 2 | 0 | 0 | 0 | 1 | 4
---|---|---|---|---|---
 3 | 0 | 1 | 0 | 1 | 6
---|---|---|---|---|---
 4 | 0 | 1 | 1 | 1 | 9
---|---|---|---|---|---
   | 1 | 7 | 4 | 9 |
```

```
# Example 2
rowSum = [1,3,5,3]
colSum = [1,3,9,0]
board=backtrack(rowSum, colSum)
printBoard(board, rowSum, colSum)

   | 1 | 2 | 3 | 4 |
---|---|---|---|---|---
 1 | 1 | 0 | 0 | 0 | 1
---|---|---|---|---|---
 2 | 0 | 0 | 1 | 0 | 3
---|---|---|---|---|---
 3 | 0 | 1 | 1 | 0 | 5
---|---|---|---|---|---
 4 | 0 | 0 | 1 | 0 | 3
---|---|---|---|---|---
   | 1 | 3 | 9 | 0 |

# Example 3
rowSum = [5, 7, 7, 12, 10]
colSum = [3, 10, 9, 7, 12]
board=backtrack(rowSum, colSum)
printBoard(board, rowSum, colSum)

   | 1 | 2 | 3 | 4 | 5 |
---|---|---|---|---|---|---
 1 | 0 | 0 | 0 | 0 | 1 | 5
---|---|---|---|---|---|---
 2 | 0 | 1 | 0 | 0 | 1 | 7
---|---|---|---|---|---|---
 3 | 1 | 1 | 0 | 1 | 0 | 7
---|---|---|---|---|---|---
 4 | 0 | 0 | 1 | 1 | 1 | 12
---|---|---|---|---|---|---
 5 | 0 | 1 | 1 | 0 | 1 | 10
---|---|---|---|---|---|---
   | 3 | 10| 9 | 7 | 12|
```

Task 3: Description and justification of approach (10 marks)

This part must be delivered a separate Word or PDF document, using the following format: **LastNameID_Task3.docx** or **pdf**. Make sure that it is included in same zip folder as your code.

Describe the brute-force or backtracking approaches that you implemented in Task2A or Task 2B. Your discussion must directly relate to each of these algorithms as you have used them to solve the Kakurasu 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.

Task 4: Decomposition, Variable names and Code Documentation (10 marks)

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