FIT1045 Algorithms and programming in Python, S2-2019
Assignment 3 (value 20%)
Due: Sunday 20th October 2019, 11:55 pm.

Objectives

The **objectives of this assignment** are:

- To demonstrate the ability to implement algorithms using basic data structures and operations on them.
- To gain experience in designing an algorithm for a given problem description and implementing that algorithm in Python.
- To demonstrate an understanding of complexity, and to the ability to implement algorithms of a given complexity.

Submission Procedure

1. Save your files into a zip file called yourStudentID_yourFirstName_yourLastName.zip
2. Submit your zip file containing your solution to Moodle.
3. Your assignment will not be accepted unless it is a readable zip file.

Important Note: Please ensure that you have read and understood the university's policies on plagiarism and collusion available at `http://www.monash.edu.au/students/policies/academic-integrity.html`. You will be required to agree to these policies when you submit your assignment.
A common mistake students make is to use Google to find solutions to the questions. Once you have seen a solution it is often difficult to come up with your own version. The best way to avoid making this mistake is to avoid using Google. You have been given all the tools you need in workshops. If you find you are stuck, feel free to ask for assistance on Moodle, ensuring that you do not post your code.

Marks: This assignment will be marked both by the correctness of your code and by an interview with your lab demonstrator, to assess your understanding. This means that although the quality of your code (commenting, decomposition, good variable names etc.) will not be marked directly, it will help to write clean code so that it is easier for you to understand and explain.
This assignment has a total of 20 marks and contributes to 20% of your final mark. For each day an assignment is late, the maximum achievable mark is reduced by 20% of the total. For example, if the assignment is late by 3 days (including weekends), the highest achievable mark is 70% of 20, which is 14. Assignments submitted 7 days after the due date will normally not be accepted.
Detailed marking guides can be found at the end of each task. Marks are subtracted when you are unable to explain your code via a code walk-through in the assessment interview. **Readable code is the basis of a convincing code walk-through.**

Task 1: Ternary Sorts (4 Marks)

Create a Python module called `ternary.py`. Within this module implement the following task. You may not import any other libraries or modules.

Task: Ternary Partition (4 Marks)

Write a function `ternary_partition(lst)` that partitions an unsorted list into three sections: smaller than pivot, equal to pivot, larger than pivot.

Input: an unsorted list containing one or more integers.

Output: a pair of integers that represent the 'pivot slice', where the first integer is the index of the first element that is equal to the pivot and the second integer is the index of the last integer that is equal to the pivot, plus one. (Note that 'plus one' is necessary because when slicing in Python the start is included and the end is excluded.) The pivot should be the element in the 0^{th} position in the original list. To receive full marks the original list must be partitioned; however, the list is not returned.

Examples

a) Let `lst1=[3]`. Calling `ternary_partition(lst1)` returns `(0,1)`, as after calling function `lst1=[3]`, thus the pivot section starts at 0 and ends at 1 (exclusive).

b) Let `lst2=[3,2,2,5,6,3,1,3]`. Calling `ternary_partition(lst2)` returns `(3,6)`, as after calling function `lst2==[2,2,1,3,3,3,5,6]` (or an approximation of this depending on the specifics of the implementation, for example `lst2==[2,1,2,3,3,3,6,5]` would also be valid), thus the pivot section starts at 3 and ends at 6 (exclusive).

c) Let `lst3=[1,2,3]`. Calling `ternary_partition(lst3)` returns `(0,1)`, as after calling function `lst3==[1,2,3]` (or an approximation of this depending on the specifics of the implementation), thus the pivot section starts at 0 and ends at 1 (exclusive).

To receive full marks, this function must have a complexity of $O(N)$, where `N=len(lst)`. This means sorting the list then finding the pivot's location in the list is not a viable solution.

Marking Guide (total 4 marks)

Marks are given for the correct behavior of `ternary_partition`:

(a) 1 mark for an implementation that does not change the original list and without a complexity of $O(N)$;

(b) 3 marks for an implementation that does not change the original list and has a best and worst-case complexity of $O(N)$;

(c) 4 marks for an implementation that changes the given list to reflect the partitioning (the list is changed 'in place') and that has a best and worst-case complexity of $O(N)$.

Note: marks will be deducted for including print statements or for including function calls outside of function definitions.

Task 2: Sudoku (10 Marks)

Sudoku is a logic-based combinatorial number-placement puzzle. The objective is to fill a $x^2 * x^2$ grid with digits so that each column, each row, and each of the $x \times x$ subgrids that compose the grid contain all of the digits from 1 to x^2. (For instance, a $9 * 9$ grid would contain nine $3 * 3$ subgrids.) The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution.[1]

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(a) An exemplary Sudoku puzzle ...

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

(b) ... and its solution

Create a Python module called `sudoku.py`. Within this module implement the following five tasks. You are encouraged to decompose the given tasks into additional functions. You may import `deepcopy` from `copy`. You may not import any other libraries or modules.

To assist with your implementation of the following tasks, you may use the following function `subgrid_values`. This function takes a $n \times n$ sudoku grid, a row coordinate, and a column coordinate, and returns a list containing the n values of the subgrid that the item at the coordinates belongs to.

```
def subgrid_values(grid, row, col):
    val = []
    #get dimension of inner box
    n = int(len(grid)**(0.5))
    #get starting row and starting col
    r = (row//n)*n
    c = (col//n)*n
    for i in range(r, r+n):
        for j in range(c, c+n):
            val.append(grid[i][j])
    return val
```

Part A: Get Grid (1 Mark)

Write a function `grid_from_file(file_name)` that reads in a file containing a sudoku grid and returns the contents of the file as a Python-readable table.

Input: a file name `file_name`, where the file contains n lines, and each line contains n entries separated by commas. Each entry will either be a positive integer, or the letter 'x'.

Output: a table represented as a nested list, where each entry in the orignal file is an element in the table, and all numbers have been converted to integers.

Examples

a) Calling `grid_from_file('gridA.txt')` returns:

```
[ ['x','x',1,'x'],
  [4,'x','x','x'],
  ['x','x','x',2],
  ['x',3,'x','x'] ]
```

b) Calling `grid_from_file('gridB.txt')` returns:

[1] Description adapted from Wikipedia. Read more here: `https://en.wikipedia.org/wiki/Sudoku`.

```
[ [1,'x',9,'x','x','x','x',6,'x'],
  [8,4,'x','x',1,'x','x',7,5],
  ['x','x',2,'x','x',3,'x','x',4],
  ['x','x',8,3,2,1,'x',4,7],
  ['x','x',5,'x','x','x',6,'x','x'],
  [4,2,'x',6,9,5,8,'x','x'],
  [7,'x','x',1,'x','x',4,'x','x'],
  [6,9,'x','x',8,'x','x',5,3],
  ['x',5,'x','x','x','x',7,'x',9] ]
```

Part B: Valid Entry (1 Mark)

Write a function `valid_entry(grid, num, r, c)` that determines whether a particular value can be entered at a particular location in a valid grid, while maintaining validity.

Input: a nested list `grid`, that represents an $n \times n$ sudoku grid; each item in the inner list is either an integer (example `13`), or the string `'x'`; a positive integer `num`, where $0 < num \leq n$; and two non-negative integers `r` and `c` that represent the row and column that `num` will be inserted, where $0 \leq r, c < n$. You may assume `grid[r][c]=='x'`.

Output: a boolean `True` if the insertion is valid; otherwise `False`. For the insertion to be valid, it must result in a grid that does not contain duplicate numbers in any row, any column, or any subgrid.

Examples

```
grid = [ [1,'x','x','x'],
         ['x','x','x','x'],
         ['x','x',1,'x'],
         ['x','x','x','x'] ]
```

a) Calling `valid_entry(grid, 1,1,3)` returns `True`.

b) Calling `valid_entry(grid, 1,0,3)` returns `False`, because there would be two `1`s in row 0.

c) Calling `valid_entry(grid, 1,1,2)` returns `False`, because there would be two `1`s in column 2.

d) Calling `valid_entry(grid, 1,3,3)` returns `False`, because there would be two `1`s in the bottom right $2 * 2$ subgrid.

Part C: Enter Number In Row (2 Marks)

Write a function `grids_augmented_in_row(grid,num,r)` that returns the complete list of valid augmented grids, where each grid contains *num* in row r.

Input: a nested list `grid`, that represents a *valid* $n \times n$ sudoku grid; each item in the inner list is either an integer (example `27`), or the string `'x'`; a positive integer `num`, where $0 < num \leq n$; and a non-negative integer `r`, where $0 \leq r < n$.

Output: a nested list containing all augmented sudoku grids such that each grid is valid, and each grid contains `num` in row `r`. If `num` is in row `r` in the original grid, return a list containing the original grid. If there is no way to augment the given grid to create a valid grid where `num` is in row `r`, return an empty list.

Remember that you may import `deepcopy` from `copy`.

Examples

```
lite_grid = [ [1,'x','x','x'],
              ['x','x','x','x'],
              ['x','x','x','x'],
              ['x',2,'x','x'] ]

full_grid = [ [2,'x','x','x'],
              ['x',3,2,4],
              ['x','x',4,2],
```

```
              [1,2,3,'x'] ]

grid_A = [ ['x','x',1,'x'],
           [4,'x','x','x'],
           ['x','x','x',2],
           ['x',3,'x','x'] ]
```

a) Calling `grids_augmented_in_row(lite_grid,1,0)` returns:

```
[
  #note there is already a 1 in row 0, so returns list containing original grid
  [ [1,'x','x','x'],
    ['x','x','x','x'],
    ['x','x','x','x'],
    ['x',2,'x','x'] ]
]
```

b) Calling `grids_augmented_in_row(lite_grid,1,1)` returns:

```
[
  [ [1,'x','x','x'],
    ['x','x',1,'x'], #note there is now a 1 in row 1
    ['x','x','x','x'],
    ['x',2,'x','x'] ],

  [ [1,'x','x','x'],
    ['x','x','x',1], #note there is now a 1 in row 1
    ['x','x','x','x'],
    ['x',2,'x','x'] ]
]
```

c) Calling `grids_augmented_in_row(full_grid,1,1)` returns `[]`, because there is no valid way to insert a `1` in row 1 of `full_grid`.

d) Calling `grids_augmented_in_row(grid_A,1,2)` returns:

```
[
  [ ['x','x',1,'x'],
    [4,'x','x','x'],
    [1,'x','x',2], #note there is now a 1 in row 2
    ['x',3,'x','x'] ] ,

  [ ['x','x',1,'x'],
    [4,'x','x','x'],
    ['x',1,'x',2], #note there is now a 1 in row 2
    ['x',3,'x','x'] ]
]
```

Part D: Enter Number (3 Marks)

Write a function `grids_augmented_with_number(grid,num)` that returns a list of valid $n \times n$ grids, where each grid contains n *num*s. (I.e. if given a $9 * 9$ grid, and $num = 3$, each grid returned must contain nine 3's.)

Input: a nested list `grid`, that represents a *valid* $n \times n$ sudoku grid; each item in the inner list is either an integer (example `13`), or the string `'x'`; and a positive integer `num`, where $0 < num \leq n$.

Output: a nested list containing all valid sudoku grids where each grid contains n `num`s. If there is no way to augment the given grid to create a valid sudoku grid containing n `num`s, return an empty list.

Examples

a) Calling `grids_augmented_with_number(lite_grid,1)` returns:

```
[
  #note there are now 1s in every row, column, and inner square in both grids
  [ [1,'x','x','x'],
    ['x','x',1,'x'],
    ['x',1,'x','x'],
    ['x',2,'x',1] ] ,

  [ [1,'x','x','x'],
    ['x','x','x',1],
    ['x',1,'x','x'],
    ['x',2,1,'x'] ]
]
```

b) Calling `grids_augmented_with_number(full_grid,1)` returns `[]`, because there is no valid way to modify `full_grid` so that it contains four 1s.

c) Calling `grids_augmented_with_number(grid_A,1)` returns:

```
[
  [ ['x','x',1,'x'],
    [4,1,'x','x'],
    [1,'x','x',2],
    ['x',3,'x',1] ]
]
```

Part E: Solve Sudoku (3 Marks)

Write a function `solve_sudoku(file_name)` that finds the solution for the given sudoku.

Input: a file name `file_name`, where the file contains n lines, and each line contains n entries separated by commas. Each entry will either be a positive integer, or the letter 'x'.

Output: a nested list representing a completed sudoku grid. You may assume the file given will always contain exactly one valid solution.

Note: you may find the functions that you are required to write in parts C and D useful in solving part E; however, you may find it more natural to solve part E using an alternative approach. If this is the case, note that you are not required to use parts C and D for your solution to part E.

Examples

a) Calling `solve_sudoku('gridA.txt')` returns:

```
[ [3,2,1,4],
  [4,1,2,3],
  [1,4,3,2],
  [2,3,4,1] ]
```

b) Calling `solve_sudoku('gridB.txt')` returns:

```
[ [1, 3, 9, 5, 7, 4, 2, 6, 8],
  [8, 4, 6, 9, 1, 2, 3, 7, 5],
  [5, 7, 2, 8, 6, 3, 9, 1, 4],
  [9, 6, 8, 3, 2, 1, 5, 4, 7],
  [3, 1, 5, 7, 4, 8, 6, 9, 2],
  [4, 2, 7, 6, 9, 5, 8, 3, 1],
  [7, 8, 3, 1, 5, 9, 4, 2, 6],
  [6, 9, 4, 2, 8, 7, 1, 5, 3],
  [2, 5, 1, 4, 3, 6, 7, 8, 9] ]
```

Marking Guide (total 10 marks)

Marks are given for the correct behavior of the different functions:

(a) 1 mark for `grid_from_file`.

(b) 1 mark for `valid_entry`.

(c) 2 marks for `grids_augmented_in_row`.

(d) 3 marks for `grids_augmented_with_number`.

(e) 3 marks for `solve_sudoku`.

Note: marks will be deducted for including print statements or for including function calls outside of function definitions.

Task 3: Jobs (6 Marks)

Create a Python module called `jobs.py`. Within this module implement the following two tasks. You may not import any other libraries or modules.

Part A: Salesperson (3 Marks)

You are an unscrupulous salesperson who sells devices that allow a household to use their neighbours' wi-fi for free. Due to the nature of the product you sell, it is never possible to sell to neighbours. Every day you find a new street at which to sell your devices. You are very good at your job, and know from market research exactly how much each household would be willing to pay.

Write a function `max_sales(street)` that finds the maximum amount of sales you can make for a given street.

Input: a list `street` of non-negative integers that represents the street you are visiting; each integer represents the amount of money the i^{th} household would be willing to pay; the i^{th} household is neighbours to the $i^{th}-1$ and $i^{th}+1$ household. The first and last household are *not* neighbours.

Output: an integer that is the maximum amount of sales you can expect to make on that street.

Examples

a) Calling `max_sales([2,3,10,5,6,0,4,12,6])` returns `30`, because you could sell to the first, third, fifth, and eighth house on the street, thus making a sale of $2+10+6+12=30$.

b) Calling `max_sales([])` returns `0`, because you cannot make any sales when selling to a street with no houses.

Part B: Burgerperson (3 Marks)

Your sales business was shutdown by the IEEE and you have had to get a job at a hip burger joint. This burger joint is different from most - the only ingredient used in the burgers is bread. Your job is to check the bread stacks made in the kitchen are actually burgers.

For a bread stack to be a burger it must follow two rules: first, any open piece of bread must be followed by a matching closing piece of bread; second, any open-close pair may contain a stack of bread between them, provided the stack is also a burger. There are three different kinds of bread: brioche (open brioche is `'ob'`, closed is `'cb'`), wholemeal (`'ow'`, `'cw'`), and rye (`'or'`, `'cr'`).

Write a function `is_burger(breads)` that determines whether a stack of bread is a burger.

Input: a non-empty list of strings `breads` that represents the bread stack you are checking; the possible strings are: 'ob', 'cb', 'ow', 'cw', 'or', 'cr'.

Output: a boolean, `True` if the bread stack is a burger; otherwise `False`. A bread stack is a burger if each open piece of bread is followed by a matching closed piece of bread, and there is either nothing between the opening and closing bread, or there is one or more burgers between the opening and closing bread.

Examples

a) Calling `is_burger(['ob'])` returns `False`, because the opening brioche does not have a closing brioche.

b) Calling `is_burger(['ob','cb'])` returns `True`.

c) Calling `is_burger(['ob','or','cb'])` returns `False`, because the brioche bun does not contain a valid burger.

d) Calling `is_burger(['ob','or','cr','cb'])` returns `True`.

e) Calling `is_burger(['ob','or','cb','cr'])` returns `False`, because the brioche bun does not contain a valid burger, and the rye bun does not contain a valid burger.

f) Calling `is_burger(['ob','or','cr','ob','cb','cb'])` returns `True`.

g) Calling `is_burger(['or','cr','ob','cb'])` returns `True`.

Marking Guide (total 6 marks)

Marks are given for the correct behavior of the different functions:

(a) 3 marks for `max_sales`.

(b) 3 marks for `is_burger`.

Note: marks will be deducted for including print statements or for including function calls outside of function definitions.