CS1010E Practical Exam II

Instructions:

- If you failed to submit your screen recording or failed to keep your proctoring camera, your PE will result in **ZERO mark** no matter what you submitted onto Luminus or Coursemology.
- This PE consists of **THREE** parts and each part should be submitted in a separated file. Namely `part1.py`, `part2.py` and `part3.py`. Each file is independent, e.g. you cannot expect that the answer in `part2.py` calls a function in `part1.y` that is not in `part2.y`. If you want to reuse some function from another file (%99 you don't need to), you need to copy them into that file. For example, you just need to copy the function you want to call in `part1.py` into `part2.py`.
- No mark will be given *if your code cannot run*, namely any syntax errors or crashes.
    - Your tutors will just grade what you have submitted and they will **not** fix your code.
    - Comment out any part that you do not want. Especially the test cases.
- Workable code that can produce correct answers in the given test cases will only give you *partial* marks. Only good and efficient code that can pass ALL test cases will give you full marks. Marks will be deducted if your code is unnecessarily long, hard-coded, or in poor programming style, including irrelevant code or test code.
- You must use the same function names as those in the skeleton files given. Also, you must NOT change the file names.
- You ***cannot import*** any additional packages or functions.
- Your code should be efficient and able to complete *each* example function call in this paper within 1 second.
- In all parts, you should **return** values instead of **printing** your output. In another word, you should not need any "`print()`" in this entire PE for submission.
- You should remove all your test cases before submission. If you submit more code than required, it will result in **penalty** because you code is "unnecessarily long".
- You should save an **exact** copy of your submission in your computer for submitting in Coursemology again after the PE. Any differences between Luminus and coursemology submissions will be severely penalized.
- Reminder: Any type of plagiarism like copying code and then modify will be caught, that include adding comments, changing variable names, changing the order of the lines/functions, adding useless statements, etc.

Constraints for Part 1

You cannot use the following Python ***built-it*** functions in Part 1:

- `encode, decode, replace, partition, rpartition, split, rsplit, translate, map, filter, map, count, find, join, rfind, rstrip, strip, sort, sorted, pop, index`

Also, your code cannot be too long. Namely, the function in *each* task must be fewer than 400 characters and 15 lines of code including all sub-functions that called by it.

Part 1 Parentheses

We use a lot of arithmetic expressions every day and there are always parentheses in them.

Part 1 Task 1 Iterative Extract Parentheses (10 marks)

Given a string `s`, write a function `extractParenthesesI(s)` to extract all the parentheses and return a string of parentheses using iterations only. You do not have to care if the string `s` is a correct arithmetic expression or not.

```
>>> print(extractParenthesesI('(1+Y)*(3+(X-5))'))
()(())
```

Part 1 Task 2 Recursive Extract Parentheses (10 marks)

Write a ***recursion*** version of the function `extractParenthesesR(s)` with the same functionality in Part 1 Task 1. However, you cannot use any loops or list comprehension in this task.

```
>>> print(extractParenthesesR('(1+Y)*(3+(X-5))'))
()(())
```

Part 1 Task 3 Check Balanced Parentheses (15 marks)

Follow the normal mathematic conventions, write a function `cbp(s)` (`cbp` = check balanced parentheses) to check if an expression has the correct parentheses balanced.

```
>>> print(cbp('(1+2)*(3+(4-5))'))
True
>>> print(cbp('(1+2)*(3+(4-5)))'))
False
>>> print(cbp('(1+2)-Y*X+(3+(4-Z)'))
False
```

Part 2 Premium Burgers

Remember the "`BurgerPrice`" problem in our assignments and tutorials? We are solving a similar problem here. Let's introduce our newly open burger shop, *Ten Ten Premium Burger Café*! We are selling some extremely good selections here. In order to provide the best dining experience, here are some ***new*** rules of our burgers.

- The bun `'B'` is free, namely $0.
- All the prices for each individual ingredient are in integer (dollars) now.
- We will give you the ingredients in a dictionary excluding the bun. E.g if '`C`' stands for cheese, '`V`' stands for veggie, '`P`' stands for patty and '`A`' stands for abalone slices (what!?). Then the prices will be given as

```
priceList0 = {'C':1,'V':3,'P':11,'A':31}
```

- If you order the ingredient by their prices in ascending order, that each price is at least doubled as the previous price. Namely, the price for `'P'` is at least 6 because the previous price for `'V'` is 3. However, you cannot assume the dictionary given is sorted.
- Each ingredient can only happen once in each burger except the bun, namely, you cannot have a burger `'BPCCB'`.
- The burger will be only sandwiched by exactly two pieces of buns, one at the start and one at the end. Namely, you cannot have a burger `'BPBCB'`.

Task 1 (30 marks)

Now, in your packet, you have `$m` and you want to enjoy the most expensive burger you can buy from our café! Write a function `buyMaxBurger(priceList,m)` with a given `pricelist` as above and an integer `m`, return a tuple that contains the most expensive burger that you can make and the change you left after buying the burger with `$m`. Note that if you cannot afford any ingredient, you will not even get the two buns with no ingredient! Your function `buyMaxBurger()` will return a tuple that contains the first item as the most expensive burger string that you can get, and the second item is how much money you left after you paid for the burger.

```
>>> print(buyMaxBurger(priceList0,0))
('', 0)
>>> print(buyMaxBurger(priceList0,26))
('BPVCB', 11)
>>> priceList1 = {'C':1,'W':2,'I':4,'T':9,'O':20,'V':41,'S':85}
>>> print(buyMaxBurger(priceList1,55))
('BCTIB', 1)
```

(Note that `pricelist1` includes the most expensive food, **S**affron, Ca**v**iar, **O**ysters, White **T**ruffle, **I**berico ham and **W**agyu beef!)

Hint: You can be *greedy* to construct your burger. Try to include the most expensive ingredient first into your "output" burger!

Part 4 Game of Life (GOF)

In this part, you are given the code that we used before:

- `create_zero_matrix(n,m)` # create a 2D array of `n` rows and `m` columns that contains zeros
- `m_tight_print(m)` # Tight printing the 2D array `m`
- `m_tight_print_gof(m)` # Similar to above but it prints spaces and "#" instead for better visual effects

The Game of Life is not your typical computer game. It is a cellular automaton, and was invented by Cambridge mathematician John Conway. It consists of a collection of cells which, based on a few mathematical rules, can live, die or multiply. Depending on the initial conditions, the cells form various patterns throughout the course of the game.

Rules

For a space that is populated (live) in the last step:

Each cell with one or no neighbors dies, as if by solitude.

Each cell with four or more neighbors dies, as if by overpopulation.

Each cell with two or three neighbors survives.

For a space that is empty or unpopulated

Each cell with three neighbors becomes populated.

The Game

The game will start with an initial `n` rows x `m` column board/2D array, with zero as an empty cell and 1 as an occupied/populated cell. The board is a 2D array created by `create_zero_matrix(n,m)`. And you can add initial occupied cells by the function `addCells(board,cell_list)`. In which the array `cell_list` is a list of tuples with the `(i,j)` index of an occupied cells. E.g. if the cell_list is equal to the following list (it is named as "`very_long_house`" in the skeleton file.):

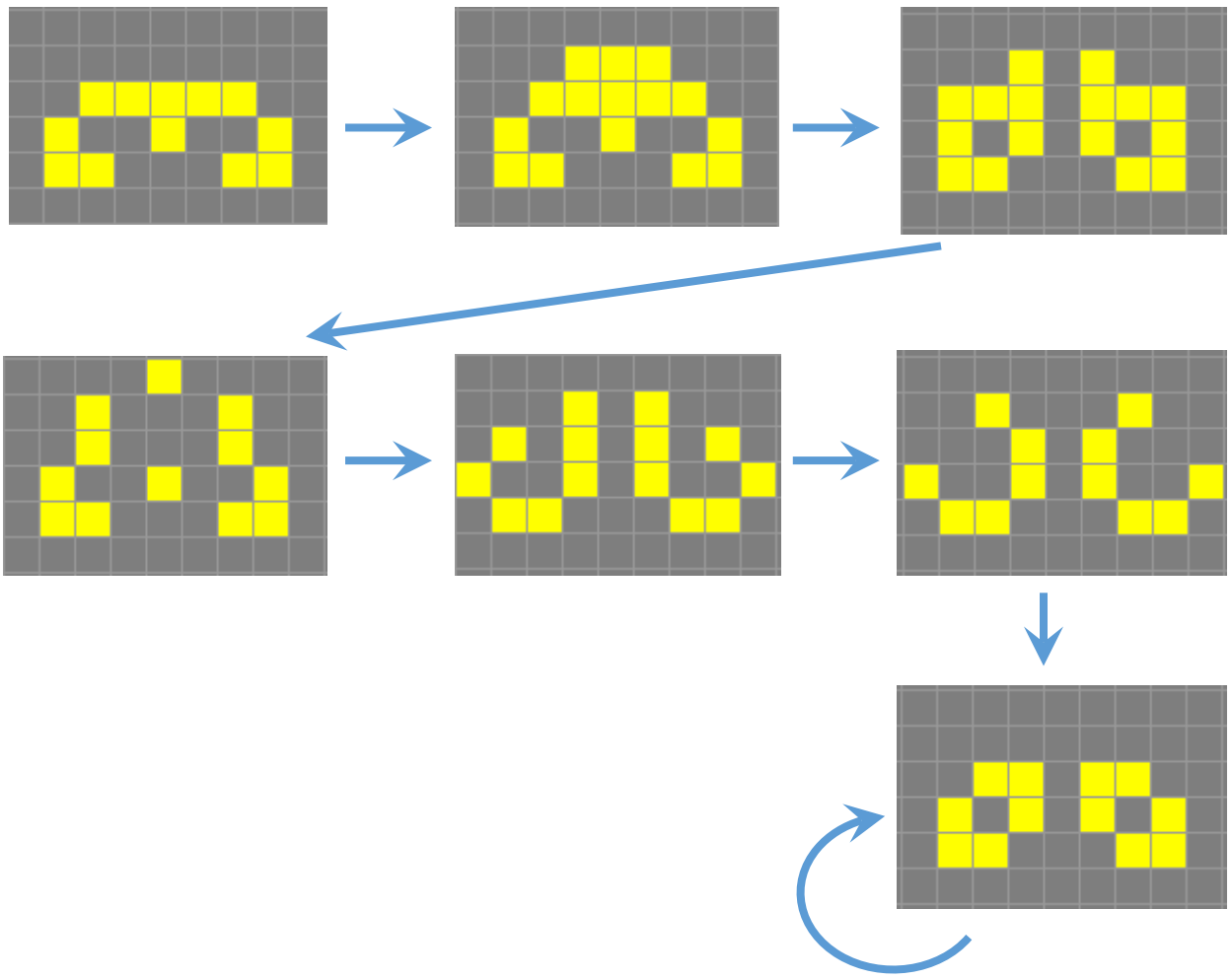
```
[(5,6),(5,7),(5,8),(5,9),(5,10),(6,5),(6,8),(6,11),(7,5),(7,6),(7,10),(7,11)]
```

Then the 2D array board will be modified according as the following with `m_Tight_print(board)` and `m_Tight_print_gof(board)`.

```
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000001111100000
0000010010010000
0000011000110000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

```
  #####
 #  #  #
 ##   ##
```

And when the game starts, all the cells will follow the ***rules*** above and evolves. For each *step*, each cell will follow the rules above to determine if that cell survives or not. And note that this is depending on *the neighbors of the last step*. Sometime the board will be "stabilized" (frozen) after some steps, but sometimes it will not. The example below is the example of `very_long_house` that will be stabilized after 6 steps.



Details of "The Neighbors of the Last Steps".

We will clarify more about what does it mean by each cell of the next step is determined by the neighbors of the last step. Here is a very simple example of a 2 x 3 grid. Let's the initial board at Step 0 be `[[0,1,0],[1,1,1]]` as below:

```
010
111
```

For the cell at (row = 0, column = 0), it has 3 neighbors in Step 0. Therefore, the cell at (0,0) will be 1 (populated) in Step 1 following the rules above

However, for cell at (0,1), it has *THREE* neighbors in Step 0 that will make it survive in Step 1. We won't count it as having FOUR neighbors because cell (0,0) becomes 1 in Step 1. The neighbor counts for cell (0,1) is *three* in Step 0, not Step 1. That's what we called "depending on ***the neighbors of the last step***."

Part 3 Task 1 `addCells()` (5 marks)

Given a 2D array `board`, implement the function `addCells(board,cell_list)` that takes in a 2D array `board` and a list of tuples `cell_list`. Each tuple in the `cell_list` contains the row and column indices for a cell to be populated. This function will add on more cells to the board and the function will ***modify*** the input `board` array directly and return `None`. You can assume the input is always a list of a tuple of two integers. But there is no limitations to the integers. If the integers are out of range, you can just ignore those out-of-range pairs and it should not cause an error or exception.

```
>>> board0 = create_zero_matrix(2,3)
>>> addCells(board0,[(1,2),(1,1),(1,0)])
>>> m_tight_print(board0)
000
111
>>> addCells(board0,[(0,1),(1,1)])
>>> m_tight_print_gof(board0)
 #
###
```

Part 3 Task 2 GOF (30 marks)

Write a function `gof(board)` and ***return*** a 2D array of the board in the next step. You **should not modify** the input `board`.

Sample outputs for Part 3

```
Test Case: Vanish
Initial board:
000
010
000
After one step:
000
000
000

Test Case: Grow
Initial board:
01
11
After one step:
11
11

Test Case: Blink
Initial board:
010
010
010
Step 1:
000
111
000
Step 2:
010
010
010
Step 3:
000
111
000
```

```
Test Case: very_long_house
Initial board:
0000000000
0000000000
0001111100
0010010010
0011000110
Step 1:

    ###
   #####
  #  #  #
  ##   ##
Step 2:
     #
    #   #
    #   #
  #  #  #
  ##   ##
Step 3:

    # #
  ### ###
  # # # #
  ##   ##
Step 4:

    # #
  # # # #
 #  # #  #
  ##   ##
Step 5:

   #   #
    # #
 #  # #  #
  ##   ##
Step 6:


   ## ##
  # # # #
  ##   ##
The board after n > 6 steps with
0 and 1's.
0000000000
0000000000
0001101100
0010101010
0011000110
```