IT5001 Practical Exam 2 Time allowed: 2 hrs

Instructions:

- Your code will be graded and run on Python 3.12.
- You *cannot import* any packages or functions. You also cannot use any decorators. However, you are allowed to write extra functions to structure your code better (except Part 1). Do remember to copy them over to the specific part(s) they are supporting.
- You can use any built-in functions without importing packages.
- No marks will be given *if your code cannot run*, namely if it causes any syntax errors or program crashes.
    - We will just grade what you have submitted and we will not fix your code.
    - Comment out any lines of code that you do not want us to grade.
    - Please do not write any "comment" with syntax errors. We used to have students put 'Task 1' in the code without any '#', which caused the code to crash.
- We have not put any test cases in Coursemology yet. Namely, if you see your submission is "all green", it does not mean your answer is perfect and 100% correct.
- Working code that produces correct answers in public test cases DOES NOT automatically mean you will get full marks. Your code must be good and efficient and pass ALL hidden test cases for you to obtain full marks. Marks will be deducted if your code is *unnecessarily long, hard-coded, in a poor programming style, or includes irrelevant code*.
- You should either delete all test cases or comment them out (e.g. by adding # before each line) in your code. Submitting more (uncommented) code than needed will result in a penalty as your code is "unnecessarily long".
- The code for each ***part*** should be submitted separately. Each part is also "independent': if you want to reuse some function(s) from another part, you will need to copy them (and their supporting functions, if any) into the part you are working on. However, any redundant functions for the part (albeit from a different part) will be deemed as "irrelevant code".
- You must use the same function name as specified in the question. You must also use the same function signature and input parameters without adding any new parameters.
- You are not allowed to mutate the input arguments.
- You should only *return* the output instead of *printing*. Therefore, you should not have any "`print()`" statement in your submission.
- Reminder: Any kind of plagiarism such as copying code with modifications will be caught. This includes adding comments, changing variable names, changing the order of lines/functions, adding useless statements, etc.
- You are not allowed to use any generative AI tools such as ChatGPT in the assessment.
- Please be reminded that you are not allowed to use any global variable

Part 1 BMI Category (5 + 15 + 15 + 10 = 45 marks)

In this part (Part 1), there are some rules:

- For each *task*, you can only write **one** function.
- In each task, you **cannot** call any function from other tasks. E.g. you cannot call the function from Task 1 in your answer in Task 3. With **one exception**: you can call the function from Task 0 in Tasks 1 to 3.
- There are four tasks in this part. You must do all of them.

BMI Category

Body Mass Index (BMI) is a numerical measure that helps assess whether a person's weight is healthy relative to their height. It's calculated using the formula:

$$BMI = \frac{weight\ in\ kg}{(height\ in\ metre)^2}$$

For example, if someone's weight is 75 kg with height 1.8m. His BMI will be

$$\frac{75}{1.8 \times 1.8}$$

Which is approximately 23.15. It is ok to assume that the height is not zero.

Based on BMI, we can broadly categorize a person into 4 groups, represented by the numbers 1 to 4:

1. Underweight: BMI < 18.5
2. Healthy Weight: 18.5 ≤ BMI < 25
3. Overweight: 25 ≤ BMI < 30
4. Obese: 30 ≤ BMI

In this part, you will be given a list of tuples with weight in kg and height in meter for several persons. For example,

```
[(75, 1.8), (60, 1.5), (80, 1.2)]
```

That means there are three persons with:

- Person A with weight 75 kg and height 1.8m
- Person B with weight 60 kg and height 1.5m
- Person C with weight 80 kg and height 1.2m

Your job is to categorize them into the FOUR categories above like this:

- Person A is in Category **2** (with BMI = 23.15)
- Person B is in Category **3** (with BMI = 26.67)
- Person C is in Category **4** (with BMI = 55.56)

Task 0 BMI Calculation (5 marks)

Write a function `BMI(w,h)` to compute the BMI of a person with weight `w` kg and height `h` m. Your function should return a floating point value instead of printing it. Here are some sample output:

```
>>> print(BMI(75,1.8))
23.148148148148145
>>> print(BMI(60,1.5))
26.666666666666668
>>> print(BMI(80,1.2))
55.55555555555556
```

Task 1 BMI Test (Iterative) (15 marks)

Write an ***iterative*** function `BMI_test_i(data)` to take the input list above and return a list of integers that indicate corresponding BMI categories. Your function must use for-loops or while-loops and your function cannot be recursive. Here are some sample output:

```
>>> print(BMI_test_i([(75,1.8),(60,1.5),(80,1.2)]))
[2, 3, 4]
>>> print(BMI_test_i([(199, 1.75), (162, 1.46), (56, 1.97), (171, 1.2), (81, 1.8)]))
[4, 4, 1, 4, 3]
```

Task 2 BMI Test (Recursive Version) (15 marks)

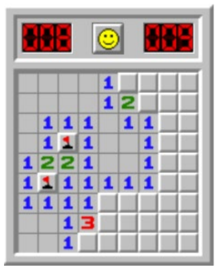
Write a ***recursive*** function `BMI_test_r(data)` with the same functionality of `BMI_test_i(data)` in the previous Task 1. Your function should be purely recursive and you cannot use any loops or list comprehension in this task.

Task 3 BMI Test (One-line Version) (10 marks)

Write a ***one-line*** function `BMI_test_o(data)` with the same functionality of `BMI_test_i(data)` in the previous task. "One-line" means that your function body must have only one return line like the below, without any semi-colon or calling other functions from Task 1 or Task 2. Additionally, your code cannot be recursive, and your function cannot exceed 150 characters including spaces.

```
def BMI_test_o(data):
    return #some value that is only one line
```

Part 2 Minesweeper (10 + 10 + 15 = 35 marks)



I hope you have played the popular game Minesweeper. However, even if you have never played the game before, we will explain it here.

You have a map of `r` rows and `c` columns with `r`, `c` > 0. In the map, there are a few cells with dangerous explosive *mines* (represented with stars `'*'`), while the remaining cells are *safe* (represented with zeroes).

Here is a sample 6x11 map with 3 mines in it.

```
00000000000
00*00000000
00000000000
000*0000000
000000*0000
00000000000
```

We provide you with a function `m_tight_print()` to print out a nicer format of the map for you to see. However, you do not need to submit that function.

Task 1 Hidden Game Creation

Write a function `gen_board(r,c, mines)` to create a 2D matrix that represents a map of `r` rows by `c` columns. The variable `mines` is a tuple of tuples, representing (row, column) positions of the mines. You can assume there are no two mines with the same location.

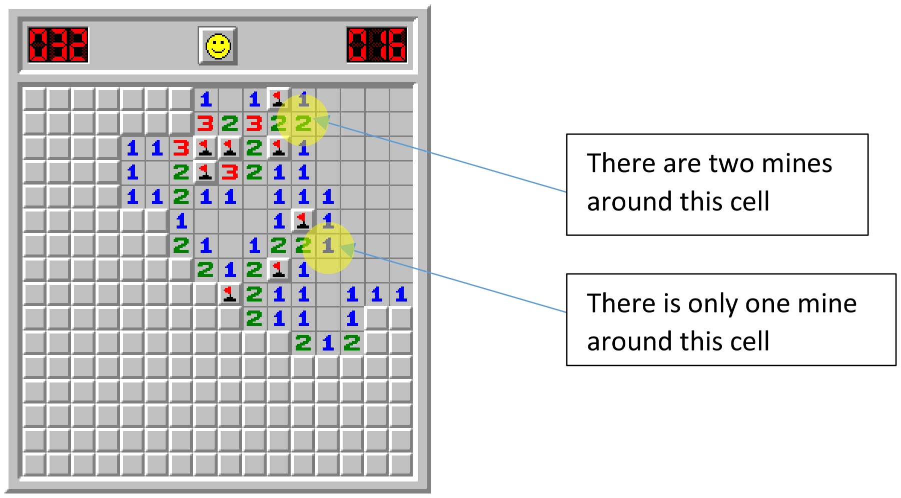
For the sample map above, we can use our function like this:

```
>>> pprint(gen_board(6,11,((4,6),(1,2),(3,3))))
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, '*', 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, '*', 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, '*', 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

>>> m_tight_print(gen_board(6,11,((4,6),(1,2),(3,3))))
00000000000
00*00000000
00000000000
000*0000000
000000*0000
00000000000
```

Task 2 Hint Data

Usually when we play Minesweeper, the map will look like the image below. You can see there are some integers on the map, which are the game *hints*. If a cell has an integer `n` in it, it means that there are `n` mines *around* the cell. “Around the cell” means any of the eight cells surrounding that cell (above, below, left, right, and diagonally: upper-left, upper-right, lower-left, lower-right) plus that central cell itself.


There are two mines around this cell
There is only one mine around this cell

Write a function `hint_board(r,c,mines)` to generate a hint map that records the number of mines around each cell. For the sample map in Task 1, your hint map should look like the following.

```
>>> pprint(hint_board(6,11,((4,6),(1,2),(3,3))))
[[0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
 [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
 [0, 1, 2, 2, 1, 0, 0, 0, 0, 0, 0],
 [0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0],
 [0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0],
 [0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0]]

>>> m_tight_print(hint_board(6,11,((4,6),(1,2),(3,3))))
01110000000
01110000000
01221000000
00111111000
00111111000
00000111000
```

Task 3 Let’s Start Sweeping!

At the start of the Minesweeper game, all cells are unopened and we do not know if any cell is safe or a mine. A player can *open* a cell by clicking it. If the cell is a mine, the game ends because the player steps on a mine. If the cell is safe, the Minesweeper game may open up more cells for you.

Your job in this task is to write a function `play_ms(r,c,mines,moves)` to play the game. The variable `moves` is a tuple of tuples, representing (row, column) positions that the player opens one by one sequentially.

Before any move is taken, all the cells on the board are unopened. Your function must show `‘#’` for unopened cells.

```
>>> pprint(play_ms(6,11,((4,6),(1,2),(3,3)), tuple()))
[['#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#'],
 ['#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#'],
 ['#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#'],
 ['#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#'],
 ['#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#'],
 ['#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#']]
```

If a move opens a mine, the game will stopped and you should put an `‘X’` in that mine cell.

If a move opens a safe cell, two things may happen:

a. If the cell has <u>hint value > 0</u>, the game will *just* open that one cell without opening up more neighbors. Your function must show the non-zero hint value in that cell.

b. If the cell has <u>hint value == 0</u>:
   - The game will first open that cell. Your function must put an empty space character (`‘ ’`) in that cell.
   - Then the game will automatically open all *connected* safe cells whose hint values are also 0. “Connected” refers to neighboring cells (around this cell in the eight directions) and transitively the neighboring cells of those neighbors … and so on, as long as their hint value is 0.
   - However, the revealing will stop when it encounters cells with hint values > 0. The game will *open* these “border” cells and show their hint values on the board, but will not continue opening their neighbors.
   - Your function must open all connected safe cells up to the borders according to the above, showing the hint value if non-zero, and an empty space character if the hint value is 0.

For example, after the player clicked at the `(0, 0)` position, the board will look like the following.

```
>>> m_tight_print(play_ms(6,11,((4,6),(1,2),(3,3)), ((0,0),)))
 1#########
 1#########
 12########
  1########
  1111#####
     1#####
```

The following sample output shows the board after *three* moves.

```
m_tight_print(play_ms(6,11,((4,6),(1,2),(3,3)), ((0,0),(0,2),(0,3))))
 111#######
 1#########
 12########
  1########
  1111#####
     1#####
```

If a move opens a mine, the game will stop immediately and it will not continue executing any remaining moves. Your function must show an "`X`" in the cell to indicate that you set off a mine. Here is an example.

```
>>> m_tight_print(play_ms(6,11,((4,6),(1,2),(3,3)), ((0,0),(0,2),(0,3),(0,4))))
 111
 1#1
 1221
  1#1111
  1111#1
     1#1

>>> m_tight_print(play_ms(6,11,((4,6),(1,2),(3,3)), ((0,0),(0,2),(1,2),(0,4))))
 11########
 1X########
 12########
  1########
  1111#####
     1#####
```

Part 3 Medical Research Tests (20 marks)

In a research hospital, a series of medical tests is scheduled for a group of patients. Each test record is uniquely identified by a number from 1 to `n`, where `n` is the total number of tests. The hospital schedules the tests in a specific order, which can be represented as a sequence of the test record numbers. For example, a schedule for `n` = 7 tests could be:

```
3, 5, 6, 4, 2, 1, 7
```

However, due to a computer virus attack, some of the test records are deleted, leaving only a few tests remaining in the corrupted schedule, perhaps like this:

```
6, 4, 2, 1
```

If we know that `n` is 7, we can infer the missing test numbers (3, 5, and 7). But to recover the original schedule, we still need to know their positioning in the sequence.

The hospital administrator then explains that if you list all possible sequences of the `n` tests in *lexicographical order*, the original test schedule is the first sequence in that list which contains the remaining (undeleted) tests in the same ordering as the corrupted schedule. Your task is to reconstruct the original test schedule.

Lexicographical Order

If you already understand what the question is asking, you can skip this part and go to the **Task** section directly. Here is a further illustration.

Lexicographical order (a.k.a. "dictionary" order) is mentioned in Lecture 2. For example, if `n = 3`, all possible sequences ("permutations") of 1, 2 and 3 will be:

```
(1,3,2), (1,2,3), (2,3,1), (3,1,2), (2,1,3), (3,2,1)
```

In the above, they are not in lexicographical order. The correct lexicographical order is:

```
(1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1)
```

If the incomplete sequence is "`1, 3`", there are three possible ways to add the missing `2`:

- `(2,1,3)`, or
- `(1,2,3)`, or
- `(1,3,2)`.

Among these possible sequences, `(1,2,3)` is the one that appears first (earliest) in the lexicographically ordered list of permutations. Therefore, `(1,2,3)` should be the correct original test schedule.

Task

Write a function `original_schedule(n,reduced_schedule)` where `n` is the original number of tests in your schedule and `reduced_schedule` is a tuple representing your corrupted schedule after some of the tests are deleted. You can assume `n` >= 0. Your function will return a tuple representing the correct original test schedule, i.e. the first permutation of tests that could contain the remaining tests in order. Here are some example output.

```
>>> print(original_schedule(7,(6,4,2,1)))
(3, 5, 6, 4, 2, 1, 7)
>>> print(original_schedule(5,(1,4,2)))
(1, 3, 4, 2, 5)
>>> print(original_schedule(3,(1,2)))
(1, 2, 3)
```

Template and Test Cases (Copy and paste each of the following into ONE .py file.)

```
###############################################################
#Part 1
def BMI(w,h):
    return 0

def BMI_test_i(data):
    return []

def BMI_test_r(data):
    return []

def BMI_test_o(data):
    return []

#Uncomment the following code for testing.
#But your submission should not include any test cases
#print(BMI_test_i([(75,1.8),(60,1.5),(80,1.2)]))  #[2, 3, 4]
#print(BMI_test_i([(199, 1.75), (162, 1.46), (56, 1.97), (171, 1.2), (81, 1.8)]))
#[4, 4, 1, 4, 3]

###############################################################
#Part 2
def m_tight_print(m): # You should not submit this function
    for row in m:
        print(''.join(map(str,row)))

def gen_board(r,c,mines):
    return []

def hint_board(r,c,mines):
    return []

def play_ms(r,c,mines,moves):
    return []

#Uncomment the following code for testing.
#But your submission should not include any test cases
#pprint(gen_board(6,11,((4,6),(1,2),(3,3))))
#m_tight_print(gen_board(6,11,((4,6),(1,2),(3,3))))
#m_tight_print(hint_board(6,11,((4,6),(1,2),(3,3))))
#m_tight_print(play_ms(6,11,((4,6),(1,2),(3,3)), ((0,0),)))
#m_tight_print(play_ms(6,11,((4,6),(1,2),(3,3)), ((0,0),(0,2),(0,3),(0,4))))
#m_tight_print(play_ms(6,11,((4,6),(1,2),(3,3)), ((0,0),(0,2),(1,2),(0,4))))

###############################################################
#Part 3
def original_schedule(n,reduced_schedule):
    return []

#Uncomment the following code for testing.
#But your submission should not include any test cases
#print(original_schedule(7,(6,4,2,1)))   #(3, 5, 6, 4, 2, 1, 7)
#print(original_schedule(5,(1,4,2)))     #(1, 3, 4, 2, 5)
#print(original_schedule(3,(1,2)))       #(1, 2, 3)
```