IT5001 Practical Exam

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 Coursemology.
- This is a test, not an assignment. The public test cases are for you to make sure that your function interface is correct, but NOT for you to verify answers. You think of some test cases to verify your answers like other written tests or quizzes. We will NOT grade you on public cases but only on private and evaluation cases.
    - Namely, you may still get zero mark even if you passed all public test cases.
- 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, 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.
- 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.
- If the function input is mutable, you should not modify the input.
- You ***cannot import*** any additional packages or functions. The only exception is that you can import the math package in Part 3.
- 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 can access the pdf files in coursemology only, but not the rest of the features such as video and forums.
    - And you cannot access other websites or documents online.
- 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 and Part 2

You cannot use the following Python ***built-it*** string/list functions in Part 1 and Part 2:

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

Or any sort functions and list comprehensions. 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.

And you cannot import any other library.

Part 1 Digit Product (32 marks)

Consider a positive integer x. multiply its nonzero digits and you get another integer y. Let's call this process "digit product". For example, we have x=808. Multiplying 8 and 8, we arrive at 64.

In this part, you can assume the input x >=10.

Before we started, you can reuse the functions in this part in Tasks 3 and 4. However, picking the wrong function to be reused is considered as wrong also.

Task 1 (8 marks)

Write an ***iterative*** version of the function `digitProductI(x)` to compute digit product of x. Here is some sample outputs.

```
>>> print(digitProductI(808))
64
>>> print(digitProductI(2**200-1))
3931840601848005869017497600000
```

Task 2 (8 marks)

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

Task 3 (8 marks)

Repeating this process on a number x again and again, you will arrive at a single digit between 1 and 9 eventually. For example:

$$808 \rightarrow 64 \rightarrow 24 \rightarrow 8$$

Write an ***iterative*** version of the function `finalDPI(x)` to compute the final digit product of a number x.

```
>>> print(finalDPI(808))
8
>>> print(finalDPI(2**200-1))
2
```

Task 4 (8 marks)

Write a ***recursive*** version of the function `finalDPR(x)` with the same functionality in Part 1 Task 3. However, you cannot use any loops or list comprehension in this task.

Part 2 Thickening and Thinning of a Black and Write ASCII Image (38 marks)

In this part, we will use a 2D array to store a black and write image. For a matrix m with r rows x c columns, the entry `m[i][j]` is considered as blank when it contains a dot, i.e. "`.`" and it is considered as filled with a pound sign "`#`". Similar to our materials, we can use `mTightPrint()` to print such an image like the following.

```
>>> pprint(m)
[['.', '.', '.', '.', '.'],
 ['.', '.', '.', '#', '.'],
 ['.', '.', '.', '.', '.']]
>>> mTightPrint(m)
.....
...#.
.....
```

Task 1 Create a 2D Array Filled with a Character (8 marks)

Write a function `createFilledMatrix(r,c,char)` that will return a 2D array with `r` rows and `c` columns filled with the character `char`.

```
>>> m = createFilledMatrix(4,6,'$')
>>> m[1][3] = 'K'
>>> pprint(m)
[['$', '$', '$', '$', '$', '$'],
 ['$', '$', '$', 'K', '$', '$'],
 ['$', '$', '$', '$', '$', '$'],
 ['$', '$', '$', '$', '$', '$']]
```

Task 2 Thickening (15 marks)

Remember that we treat '`.`' as blank and '`#`' as filled? Here is an example of a picture. (You can call it a stickman or the Chinese character "big", pronounced as "dai")

```
>>> mTightPrint(pic)
............
............
......#.....
......#.....
...#######..
......#.....
......#.....
......#.....
.....#.#....
....#...#...
...#.....#..
............
............
```

Thickening a picture is to make each pixel or line "thicker". For example, the thickening of the above picture will be like the picture below on the left. And we can thicken the result once more shown on the right.

```
>>> tpic = thickening(pic)
>>> mTightPrint(tpic)
............
.....###....
.....###....
..#########.
..#########.
..#########.
.....###....
....#####...
...#######..
..#########.
..####.####.
..###...###.
............
```

```
>>> ttpic = thickening(tpic)
>>> mTightPrint(ttpic)
....#####...
....#####...
.###########
.###########
.###########
.###########
.###########
..#########.
.###########
.###########
.###########
.###########
.#####.#####
```

The rule of thickening is very simple, for every **filled** pixel in the original picture (i.e. the `'#'`), fill the surrounding eight pixel in the new picture. Here is an example below. You do not need to produce the red color. The red color is just to help you to see which the additional pixels are. You do not need to enlarge the size of the picture if some additional pixels are out of bound. Namely, you can just ignore those which have "spilled out".

```
>>> mTightPrint(m2)
..........
..........
...#......
.......#..
..........
..........
>>> mTightPrint(thickening(m2))
..........
..###.....
..###.###.
..###.###.
......###.
..........
```

You task is, to write a function `thickening(m)` that takes in a 2D array m with `'.'` and `'#'` only, and return a copy of the picture that is "thickened" as another 2D array. Your function should NOT modify the input m.

Task 3 Thinning (15 marks)

Thinning is kind of the opposite operation of thickening. Here are some examples starting from the twice thickened picture `ttpic` from the previous task.

```
>>> pic2 = thinning(ttpic)
>>> mTightPrint(pic2)
............
.....###....
.....###....
..#########.
..#########.
..#########.
...#######..
...#######..
...#######..
..#########.
..#########.
..###...###.
............
```

```
>>> pic3 = thinning(pic2)
>>> mTightPrint(pic3)
............
............
......#.....
......#.....
...#######..
....#####...
....#####...
....#####...
....#####...
....#####...
...#.....#..
............
............
```

```
>>> pic4 = thinning(pic3)
>>> mTightPrint(pic4)
............
............
............
............
............
.....###....
.....###....
.....###....
.....###....
............
............
............
............
```

```
>>> pic5 = thinning(pic4)
>>> mTightPrint(pic5)
............
............
............
............
............
............
......#.....
......#.....
............
............
............
............
............
```

The simple rule for thinning is: For a filled pixel '#' in the original picture, it will disappear in the output. if it does not have eight filled neighbors around it in the original picture. Your task is to implement this `thinning(m)` function.

Part 3 Smallest Multiple (30 marks)

In this part, you can import any functions from the math package only.

We all learned LCM in our primary school when we were young.

In arithmetic and number theory, the **least common multiple (LCM)** (a.k.a. lowest common multiple, or smallest common multiple) of two integers a and b, usually denoted by lcm(a, b), is the smallest positive integer that is divisible by both a and b. We can assume that a and b are both greater than 0 in this part.

For more than two numbers, we can also apply the same concept. Given a set of integers S, lcm(S) is the smallest positive integer that is a multiple of every integer in S. For example, if S = {1, 2, 3, 4}, the LCM of S is 12. Since 12 is the smallest positive number that can be divided by 1,2,3 and 4.

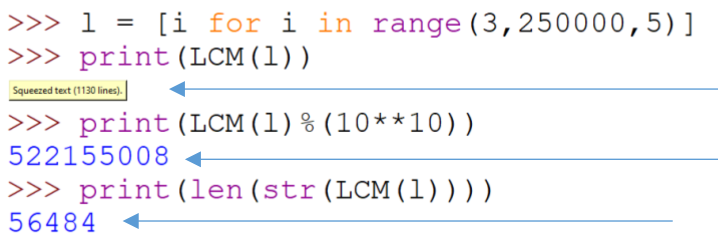
Task 1 (15 marks)

Write a function `lcm(S)` to compute the LCM of S that is a list of integers. You can assume that S contains at least one integer. Here are some example outputs:

```
>>> print(LCM([2,3,4,2]))
12
>>> print(LCM([2,3,7,11,19,23,29,4]))
11709852
>>> print(LCM([399,772,163,959,242]))
832307365428
```

Task 2 (15 marks)

Your LCM function is able to compute a very large set of integers within 1 second, e.g. with the size of S > 50000.



```
>>> l = [i for i in range(3,250000,5)]
>>> print(LCM(l))
Squeezed text (1130 lines).
>>> print(LCM(l)%(10**10))
522155008
>>> print(len(str(LCM(l))))
56484
```

The answer 1139 lines long

The last 10 digits of the answer

The answer has 56484 digits

- End of the Paper -