IT5001 Practical Exam

Instructions:

- You have to remember your Coursemology user login and password. We will not compensate anytime if you need time to retrieve them
- It is NOT an assignment. So we will merely grade your code depending on your correctness. Also, no mark will be given for hard-coding
- You ***cannot import*** any packages or functions or you will get zero marks. You also cannot use any decorators.
- This PE consists of **THREE** parts: The code for each part should be copied separately over to Coursemology. 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.
- You are allowed to write extra functions to structure your code better. However, remember to copy them over to Coursemology and into the specific part(s) they are supporting.
- 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.
- Working code that can produce correct answers in public test cases will only give you *partial* marks. Your code must be good and efficient and pass ALL 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 or test code.
- By passing ALL public and private cases does not mean that you will get full marks. We have not added in all the test cases in the PE because we want to lighten up the load of submission during the PE. We will add more test cases after the PE
- You must use the same function names as those given in the code template.
- Your code should be efficient and be able to execute *each* sample function call in this document within 1 second.
- In all parts, you should **return** instead of **print** your output, so there is no need for any "`print()`" statements.
- 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".
- 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.

Part 1 Plinko (10 + 10 + 10 = 30 marks)

In this part, you cannot use any list comprehension or map/lambda functions.

In a carnival, we open a Plinko booth and let players release a ball from the top. The ball will land in a numbered slot at the bottom. Depending on which slot the ball lands in, the player will get the prize corresponding to its number. If the ball lands in 0, they will get the big prize, 1 for the medium and 2 for the small. However, the number of prizes is limited: We only have `b` big prizes, `m` medium prizes and `s` small prizes.

Our boss is very paranoid: He is afraid of people suing him if their ball lands in a slot but its corresponding prize has run out. Hence, the booth closes immediately if and when any category (i.e. big, medium or small) runs out of prizes. Assuming no customer plays twice, we can (somehow) predict the Plinko sequence in a tuple, e.g.

`(1, 1, 2, 2, 0, 2, 0, 1, 2, 1)`

For instance, if `b=2`, `m=3` and `s=5`, the above prediction would make the booth close after serving the 7th customer (with the 0 underlined) because the big prizes would have run out, so the total number of customers served would be 7.

Your job in this part is to compute the number of customers who would be served before the booth closes, given a Plinko prediction and the number of available prizes in each category.

Task 1 Plinko Iterative Version

Write an ***iterative*** function `plinko_i(seq,b,m,s)` which **returns** the number of customers served. The input `seq` is the Plinko prediction, a **tuple** of arbitrary length containing the integers `0`, `1` and `2`. The three parameters `b`, `m` and `s` are the number of big, medium and small prizes respectively with `b, m, s >= 0`. Your function cannot be recursive in this task. Here are some example inputs and outputs:

```
>>> print(plinko_i((1, 1, 2, 2, 0, 2, 0, 1, 2, 1),2,3,5))
7
>>> print(plinko_i((0,0,0,0,0,1,0,1,2),6,2,2))
7
>>> print(plinko_i((0,1,2,0,1,2),3,3,2))
6
```

Task 2 Plinko Recursive Version

Write a ***recursive*** version `plinko_r(seq,b,m,s)` of the function in Part 1 Task 1 with the same functionality. You cannot use any loops or list comprehension in this task.

Task 3 Plinko General Version

Write a function `plinko_general(seq,prizes)` to extend the above functionality to `N` different prize categories instead of only 3 with `N > 0`. The input `seq` will now be a tuple of any length but with integers from `0` to `N-1` representing the slot numbers of each prize category. Instead of using the three variables `b`, `m` and `s`, the number of prizes in each category will be given as another input tuple `prizes`.

```
>>> print(plinko_general((0,1,2,0,1,2,0,1,2,2,2,1,1,0,1),(4,3,4)))
8
>>> print(plinko_general((0,1,3,2,1,2,3,4,5,4),(2,3,3,3,1,5)))
8
```

Part 2 Archaeologist Text Fragment Matching (20 + 10 marks)

Archaeologists recover old documents from ancient times, but these recovered documents are usually highly fragmented. One job of an archaeologist is to "glue" back the words of a document from its fragments. For example, if we found some fragments like the following:

```
'ard', 'b', 'cl', 'cke', 'ould'
```

We could join two of them up to form words like `'bard'` or `'cloud'`.

Given a file which contains fragments and string `w`, your job is to find any possible ways where two fragments in the file can be concatenated to form `w`. You can have a look at the sample input files with names starting with `'fragment'`.

```
umbre
usl
abk
lla
um
brella
```

There will be one fragment per line in the file. For example, the right textbox shows the lines of the sample file `'fragment_simple.txt'`. We can check for any ways where two fragments on the right can be concatenated to form the word `'umbrella'`:

```
>>> print(fragment('fragment_simple.txt','umbrella'))
[('um', 'brella'), ('umbre', 'lla')]
>>> print(fragment('fragment1.txt','python'))
[('pyt', 'hon')]
```

The second example above checks for ways two fragments of `'fragment1.txt'` can be concatenated to form `'python'`. Each tuple is in the correct order for the concatenation. For example, `('pyt', 'hon')` is correct but `('hon', 'pyt')` is not. You can assume all fragments and the word `w` only contain lowercase letters of the English alphabet. All fragments are unique, so there should not be any duplicate tuples in the output list. The list must also be sorted **lexicographically** by the first fragment of the tuple. Moreover, you can assume the word `w` will not be one single fragment in a file.

Task 1 Searching for Fragments

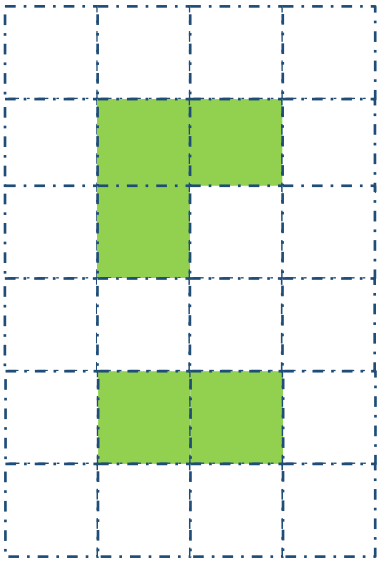
Write a function `fragment(filename,word)` that returns a list of tuples, where each tuple contains the two fragments in the file named `filename` which can be concatenated in that order to form the given `word`.

Task 2 Handling Big Files

You can only get full marks for this part if your function can handle very large input files within 1 second. For example, the computation time for each of the following function calls with large input files is less than 1 second.

```
>>> print(fragment('fragment_all2.txt','board'))
[('bo', 'ard'), ('boa', 'rd')]
>>> print(fragment('fragment_all3.txt','board'))
[('b', 'oard'), ('bo', 'ard'), ('boa', 'rd'), ('boar', 'd')]
```

Part 3 Island Perimeter (20 + 20 marks)



Imagine that our world is pixelated! Everything appears as squares, and the length of each square is 1 unit. Here is a map of some islands on the right: The solid (green) squares represent land and the hollow squares represent water.

We are interested to find the total perimeter of all the islands. For example, the total island perimeter of the right map is 14 as the rotated L-shaped island has a perimeter of 8 and the lower rectangular island has a perimeter of 6.

Task 1 Total Perimeter

You are given a map in the form of a 2D matrix, with 0 and 1 representing water and land respectively. For instance, the right map is represented by the following matrix:

```
map0 = [[0, 0, 0, 0], [0, 1, 1, 0], [0, 1, 0, 0],
        [0, 0, 0, 0], [0, 1, 1, 0], [0, 0, 0, 0]]
```

Write a function `total_perimeter(mp)` which takes in a 2D matrix `mp` as a map and returns the total perimeter of all the islands as an integer. You will need to count any island edge which touches the boundary of the map like in `map1`:

```
>>> print(total_perimeter(map0))
14
>>> map1 = [[0, 0, 1, 0, 0, 0], [0, 0, 1, 1, 1, 1], [1, 1, 0, 0, 0, 0], [0,
1, 0, 1, 1, 1], [0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 0]]
>>> print(total_perimeter(map1))
36
>> print(total_perimeter(map2a)) #given in the template in coursemology
60
```

If there is no island/land in the map, return 0.

Task 2 Maximum Perimeter

In the first example `map0`, there are two independent islands with perimeters 8 and 6 respectively. Hence, the maximum perimeter of all the islands on that map is 8. In this task, write a function `max_island_perimeter(mp)` which returns the maximum island perimeter of map `mp`.

```
001000
001111
110000
010111
010101
000100
000000
```

Note that two land squares belong to the same island if and only if they are connected horizontally or vertically, but NOT diagonally. For instance, `map1` has three islands as shown on the right, and the lower right inverted U-shaped island gives the maximum island perimeter of 14. If there is no island/land in the map, return 0.

```
>> print(max_island_perimeter(map0))
8
>>> print(max_island_perimeter(map1))
14
>>> print(max_island_perimeter(map2a))
28
```

-- End of Paper --