IT5001 Practical Exam

Instructions:

- If you failed to keep your proctoring camera, your PE will result in **ZERO mark** no matter what you submitted onto Coursemology.
- You ***cannot import*** any additional packages or functions, or you will get zero mark.
- This PE consists of **FOUR** parts and each part should be submitted (copied to) separately in Coursemology. It's better for you to organize your files into independent files. And your submission for each part is also "independent". If you want to reuse some function(s) from another file/part, you need to copy them (as well as their supporting functions) into that file/part.
- You are allowed to write extra functions to structure your code better. However, remember to submit together mentioned in the point above.
- No mark will be given *if your code cannot run*, namely any syntax errors or crashes.
    - We will just grade what you have submitted and we 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 template given.
- 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 either delete all test cases before submission or comment them out by adding # before the test cases. If you submit more code than required, it will result in **penalty** because your code is "unnecessarily long".
- Reminder: Any type of plagiarism like copying code with modifications 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 Task1* and *Part 1 Task 2*

You cannot use the following Python ***built-in*** functions/methods in Part 1 (Tasks 1 and 2):

- `encode, decode, replace, partition, rpartition, split, rsplit, translate, map, filter, join.`

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 are called by it.

You cannot use any `list`, `tuple`, `dict` or `set` and their functions.

Part 1 Name Similarity Degree (10 + 10 + 10 = 30 marks)

A new scientific research shows that two persons are more likely to be good friends if their names are similar. To measure how similar two names are, we count the number of characters that are the same and in the same positions. For example, the two names 'Brandy' and 'Flank' has a similarity of 2. If the names are not in the same length, we just align them on the left and compare up to the last letter of the shorter one. Note that two letters are the same even one of them is upper case and the other is lower case.

Task 1 Iterative Name Similarity Degree (10 marks)

Write an ***iterative*** version of the `common_char_i(name1,name2)` to compare two strings of names and return the number of same letters in the same position. In this task, you cannot use any recursion. Here is some sample output.

```
>>> print(common_char_i('Mark','Mary'))
3
>>> print(common_char_i('Brandy','Flank'))
2
>>> print(common_char_i('Larry','Clark'))
1
>>> print(common_char_i('Teddy','Andy'))
1
>>> print(common_char_i('McDonald','Andrey'))
1
```

Task 2 Recursive Name Similarity Degree (10 marks)

Write a ***recursion*** version of the function `common_char_r(name1,name2)` with the same functionality in Part 1 Task 1. However, you cannot use any loops or list comprehension in this task.

Task 3 One-liner Name Similarity Degree (10 marks)

Write a version of the function `common_char_u(name1,name2)` with the same functionality in Part 1 Task 1. However, the body of your function should be only one line and you can use any functions that were forbidden in Part 1 Tasks 1 and 2. Meaning, you can use anything including list comprehension. However, you are still not allowed to import any extra packages.

Part 2 Text Compression (20 marks)

We want to make a string shorter by replacing a word with an integer if the word appears after the first occurrence. We will replace the i^{th} occurrence of a word for i > 1.(So we will keep the first occurrence of every word.) And the i^{th} occurrence of the word will be replaced by the position j of the first occurrence of that word with position starting with 1 in the string. Note that two words are considered to be the same even if they are in different lower or upper cases. Your task is to write a function `text_compression(text)` to take in a string of text and return a compressed string as the following examples. However, there is one more catch: if your word is only *one letter*, it will not be replaced by a number. See the 'a' replacement in the examples below.

You can assume the input and your output should be a string of letters or spaces, in which, there is only one space between two words. Moreover, there will be no leading or trailing spaces.

```
>>> text7 = 'Text compression will save the world from
inefficiency Inefficiency is a blight on the world and its
humanity'
>>> print(text_compression(text7))
Text compression will save the world from inefficiency 8 is a
blight on 5 6 and its humanity
>>> text2 = 'To be or not to be'
>>> print(text_compression(text2))
To be or not 1 2
>>> text3 = 'Do you wish me a good morning or mean that it is a
good morning whether I want not or that you feel good this
morning or that it is morning to be good on'
>>> print(text_compression(text3))
Do you wish me a good morning or mean that it is a 6 7 whether I
want not 8 10 2 feel 6 this 7 8 10 11 12 7 to be 6 on
```

Part 3 ASCII Picture Pattern Matching (20 marks)

Remember ASCII pictures? An ASCII picture in Python is a 2D array (a list of list) of ASCII characters. E.g. here is an example of an ASCII picture with 15 rows and 43 columns (the variable **`pic`** given in your skeleton code.) printed by the function `mTightPrint()` given in the lecture.

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

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

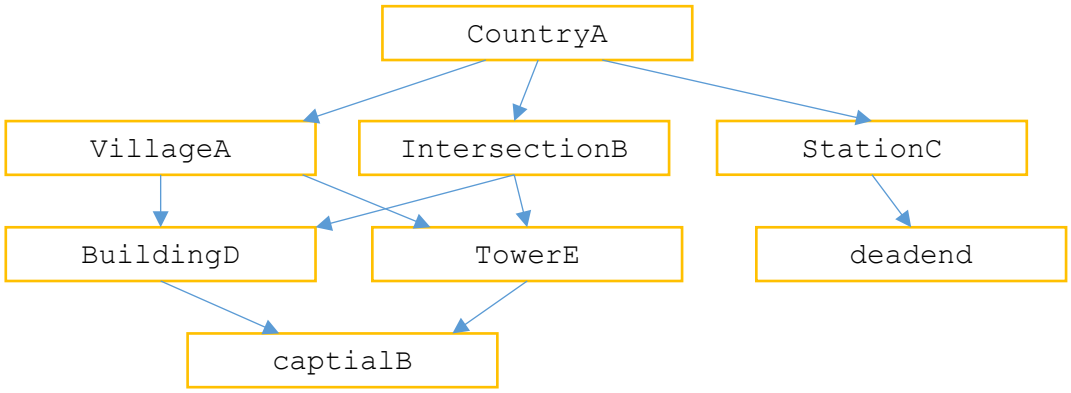
Given the bigger picture `pic1` and the smaller pattern `part1` as follows. The function `pattern_matching(pic1,part1)` should return the areas of all occurrences of `part1` in `pic1`. As a list containing tuples and each tuple will be 4 integers, namely, the smallest row index, the smallest column index, the larger row and column indices of the pattern `part1` in `pic1`. E.g. `(1,2,3,4)` states the pattern appears from row 1 to row 3 and column 2 to column 4 marked as red, and `(3,4,5,6)` indicates the green one.

```
>>> pprint(pic1)
[['.', '.', '.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '#', '.', '#', '.', '.', '.', '.', '.'],
 ['.', '.', '.', '#', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '#', '.', '#', '.', '#', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '#', '.', '.', '.', '.'],
 ['.', '.', '.', '.', '#', '.', '#', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.', '.', '.']]
>>> mTightPrint(pic1)
..........
..#.#.....
...#......
..#.#.#...
.....#....
....#.#...
..........
>>> print(part1)
['#.#', '.#.', '#.#']
>>> print(pattern_matching(pic1,part1))
[(1, 2, 3, 4), (3, 4, 5, 6)]
```

Part 4 War Strategy Prediction (30 marks)

Unfortunately, Country A is starting a war to invade Country B. Sir Nala, the defense commander of Country B, knows what are the pattern of Country A's invasion. He receives the intel of how Country A will advance their troops. The ultimate goal of Country A, of course, is to reach the capital of Country B, named as `'capitalB'`. The intel is given as a map like the following in a text file

```
CountryA VillageA IntersectionB StationC
VillageA BuildingD TowerE
IntersectionB TowerE BuildingD
StationC deadend
TowerE capitalB
BuildingD capitalB
```



Each line says that a possibility of Country A Troops advancement. For each line, the first word is the starting point of each step and the subsequent words are the possible advancement from the first word. E.g. in the above map file, the first line states that from `CountryA`, their troops can go to three places, namely, `VillageA`, `IntersectionB` and `StationC`. In the same manner, Line 2 states that their troops can advance *from* `VillageA` to either `BuildingD` or `TowerE`. Eventually, the troops will end up in either `capitalB`, or a `'deadend'` that the troops cannot move further anymore. There is not limitation of how many `'capitalB'` or `'deadend'` appearing in the map given. They can be ranging from 0 to any number.

So Sir Nala, as the defense commander of Country B, wants to know all the possible ways that Country A can reach `capitalB`. In the above example, there are 4 different ways as follows:

```
CountryA  → VillageA → BuildingD → capitalB
CountryA → VillageA → TowerE → capitalB
CountryA → IntersectionB → TowerE → capitalB
CountryA → IntersectionB → BuildingD → capitalB
```

Fortunately, Sir Nala only needs to know what is **the number of ways** instead of the details of all different ways. Write a function `strategic_count(mapfile)` to **read** in a map file as above to return the number of ways that the troops can go from `'CountryA'` to `'captialB'`. You can assume the map that:

- Does not have loops. And the map direction is not bidirectional. Namely `CountryA` can go to `VillageA` but not the other way round.
- There is a starting point of `'CountryA'`
- Every route will either end up with `'capitalB'` or `'deadend'`

Note that you will get full marks if your code works with very large cases within 1 second, e.g. the final answer > 1 million.