IT5001 Practical Exam

Instructions:

- Your code will be graded and run on **Python 3.10**.
- You ***cannot import*** any packages or functions unless some questions specifically allow you to do so. 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.
- 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 public and hidden test cases for you to obtain full marks. Marks will be deducted if it 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.**
- In general, you should **return** instead of **print** your output, unless you are specifically told to do so.
- 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.**

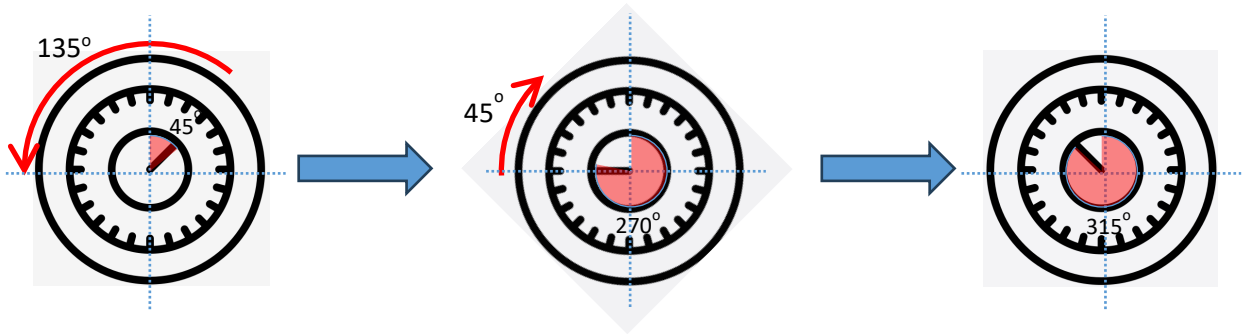
Part 1 Safe Dial (10 + 10 + 10 = 30 marks)

For all tasks in this part, **you must write only one function for each task.** In particular, for Tasks 1 and 2, you cannot use any list comprehension or map/lambda functions, as well as any of the following ***built-in*** functions:

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

In the olden days, people kept valuable items in safes. Most safes in the past used mechanical dials as their lock. To unlock a safe, one had to repeatedly turn the dial in a sequence of clockwise and/or counterclockwise turns.

We define the current state of a dial as the angle from the "north" direction to its direction, i.e. we determine the angle of the dial by starting from the north and measuring the number of degrees clockwise to the direction that the dial is aligned towards. For example, we can start with a dial at 45 degrees. Then we turn it by 135 degrees in the counterclockwise direction. After another clockwise turn of 45 degrees, the dial's state will be at 315 degrees.



Given a starting degree and a sequence of turns, your job is to compute the final state of the dial after performing the sequence of turns. The sequence of turns will be given as a list of tuples where each tuple describes the turn, giving its direction (clockwise (`'CW'`) or counterclockwise (`'CCW'`)) followed by the degree (as an integer).

Task 1 Dial Iterative Version (10 marks)

Write an ***iterative*** function `dial_i(start,seq)` which **returns** the final degree `n` of the dial. Your return value `n` should be between 0 and 359, namely, $0 \leq n \leq 359$. Your function cannot be recursive in this task.

```
>>> print(dial_i(45,[('CCW',135),('CW',45)]))
315
>>> seq1 = [('CW',40),('CW',100),('CCW',10),('CW',30),('CCW',20),('CW',180)]
>>> print(dial_i(70,seq1))
30
>>> print(dial_i(350,seq1))
310
```

Task 2 Dial Recursive Version (10 marks)

Write a ***recursive*** version `dial_r(start,seq)` of the function in Task 1, with the same functionality. Your function should be purely recursive, which means you cannot use any loops or list comprehension in this task.

Task 3 Dial One-liner Ultimate Version (10 marks)

Write the function `dial_u(start,seq)` with the same functionality as Tasks 1 and 2, but your function must only contain one statement which is a return statement, so the function body should only have one line (without any semi-colon). In this task, you may use some of the *"forbidden"* functions stated above. However, you still cannot import any packages. Your function should also be totally independent from the previous tasks:

```
# your function should look like this:
def dial_u(start,seq):
    return #your code here with only one line

# your function should NOT look like this because it is NOT independent
def dial_u(start,seq):
    return dial_i(start,seq)
```

Part 2 Prefix Matching (5 + 15 + 10 = 30 marks)

In a special research centre, its staff can only enter the centre by spelling out their passwords. Each staff is assigned a unique password. However, because there is no "enter" key to indicate that the password is finished, a password will be recognized as soon as the letters that have been spoken so far match one of the passwords in the database.

This voice authentication system poses a problem: Each password in the database cannot be a prefix of another password. Given two strings `s1` and `s2`, `s1` is a ***prefix*** of `s2` if `s1` is the initial segment of `s2`. E.g. **`'abc'`** is a prefix of **`'abc`**`de'` but `'abc'` is NOT a prefix of `'zabc'`. A string is also a prefix of itself.

You can assume all passwords are strings which are (1) NOT empty and (2) alphanumeric (i.e. contain only English alphabets from a-z, A-Z and/or numbers from 0-9).

Task 1 Prefix (5 marks)

Write a function `prefix(s1,s2)` which returns `True` if either `s1` is a prefix of `s2` ***or*** `s2` is a prefix of `s1`, otherwise, return `False`.

```
>>> print(prefix('abc','abcdz'))
True
>>> print(prefix('abc','kkkabc'))
False
>>> print(prefix('xyzabc','xyz'))
True
```

Task 2 Check all passwords (15 marks)

Your job is to make sure that there is no password that is the prefix of *another* password in the database. Given a list of passwords `seq`, write a function `check_prefix(seq)` which returns `True` if there exists at least one password that is a prefix of another password in `seq`, and return `False` otherwise.

```
>>> passwords1 = ['abc','assfs','asfjl','xy987x','12312','jxljb','11515']
>>> print(check_prefix(passwords1))
False
>>> print(check_prefix(passwords1+['abcdk']))
True
```

Task 3 Check many many many passwords (10 marks)

Your `check_prefix()` function must be able to handle very large lists within 1 second, i.e. lists of length > 40000. For example, the following code will generate a list of more than 40000 passwords. But do not submit the following code. It is for your own testing only.

```
from itertools import permutations as perm
from time import time
passwords2 = list(perm(['a','b','c','d','e','f','g','h','i']))
print(len(passwords2))

st = time()
print(check_prefix(passwords2))
print("Time: ",time()-st)
st = time()
print(check_prefix(passwords2+[passwords2[-1]*2]))
print("Time: ",time()-st)
```

And if you run the above code with the long list `passwords2`, an efficient code should give:

```
362880
False
Time:  0.16645026206970215
True
Time:  0.1503467559814453
```

Part 3 Formula 1 Checkered Flag (20 + 10 + 10 = 40 marks)

It's the racing season again! We are going to make the checkered flag like the one on the right.

Task 1 Checkered Flag (20 marks)

Write a function `checkered_flag(r,c,s)` which returns a 2D array with `r` rows and `c` columns, where `r,c` >= 0. The symbol `'#'` represents black and the space `' '` represents white. The length of each square is `s` where `s` > 0. The function should return the checkered flag as a 2D array `m` with `m[0][0] = '#'`:

```
>>> pprint(checkered_flag(5,7,2))
[['#', '#', ' ', ' ', '#', '#', ' '],
 ['#', '#', ' ', ' ', '#', '#', ' '],
 [' ', ' ', '#', '#', ' ', ' ', '#'],
 [' ', ' ', '#', '#', ' ', ' ', '#'],
 ['#', '#', ' ', ' ', '#', '#', ' ']]

>>> pprint(checkered_flag(8,10,3))
[['#', '#', '#', ' ', ' ', ' ', '#', '#', '#', ' '],
 ['#', '#', '#', ' ', ' ', ' ', '#', '#', '#', ' '],
 ['#', '#', '#', ' ', ' ', ' ', '#', '#', '#', ' '],
 [' ', ' ', ' ', '#', '#', '#', ' ', ' ', ' ', '#'],
 [' ', ' ', ' ', '#', '#', '#', ' ', ' ', ' ', '#'],
 [' ', ' ', ' ', '#', '#', '#', ' ', ' ', ' ', '#'],
 ['#', '#', '#', ' ', ' ', ' ', '#', '#', '#', ' '],
 ['#', '#', '#', ' ', ' ', ' ', '#', '#', '#', ' ']]
```

Task 2 Calculate Black Area (10 marks)

To print the flag, we want to know how much black ink we need. Each symbol `'#'` represents one unit of the black area. Therefore, the number of units of black area in a flag is equal to the total number of `'#'`s. Write a function `black_area(r,c,s)` which **returns** the number of `'#'`s in the checkered flag.

```
>>> print(black_area(5,7,2))
18
>>> print(black_area(8,10,3))
42
```

Task 3 Big Flag Black Area (10 marks)

For this task, your `black_area(r,c,s)` function should be able to handle very large flags within 1 second, e.g.

```
>>> print(black_area(21073,31171,8))
328433243
>> print(black_area(723847,1213111,114))
439053381049
```

Appendix: Code Template

You might need to fix some whitespace inconsistencies. Each part should be copied to a separate file for your submission. Do comment out or omit any **test code**.

Part 1

```
def dial_i(start,seq):
    pass

def dial_r(start,seq):
    pass

def dial_u(start,seq):
    pass
```

Part 2

```
def prefix(s1,s2):
    pass

def check_prefix(seq):
    pass

#The following code is for your testing only. Do NOT submit them
#Submitting them will suffer in *unnecessarily long code* mark deduction
'''
from itertools import permutations as perm
from time import time
passwords2 = list(perm(['a','b','c','d','e','f','g','h','i']))
print(len(passwords2))

st = time()
print(check_prefix(passwords2))
print("Time: ",time()-st)
st = time()
print(check_prefix(passwords2+[passwords2[-1]*2]))
print("Time: ",time()-st)
'''
```

Part 3

```
def checkered_flag(r,c,s):
    pass

def black_area(r,c,s):
    pass
```

-- End of Paper --