IT5001 Practical Exam

Instructions:

- Your code will be graded and run on **Python 3.12**.
- For this whole paper, you ***cannot import*** any packages or functions and you also cannot use any decorators. However, you are allowed to write extra functions to structure your code better (unless specified). 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.
    - Remove (or comment out) any lines of code that you do not want us to grade. 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" in each part.
- 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*.
- 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 "unnecessarily long 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 **`return`** your output instead of **`print()`**. And you should not include any debug or test code in your submissions.
- 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. Basically, you should write all the code from scratch.
- **You are not allowed to use any generative AI tools such as ChatGPT in this assessment.**
- Before the PE ends, remember to
    - Save your three parts in three files in your computer.
    - Submit your code to Examplify, and make sure that they are the same as your saved copies.
    - **Do not modify any more in Examplify and your saved copies before and after PE ends.**

Part 1 Flying a Drone (15 + 15 + 10 = 40 marks)

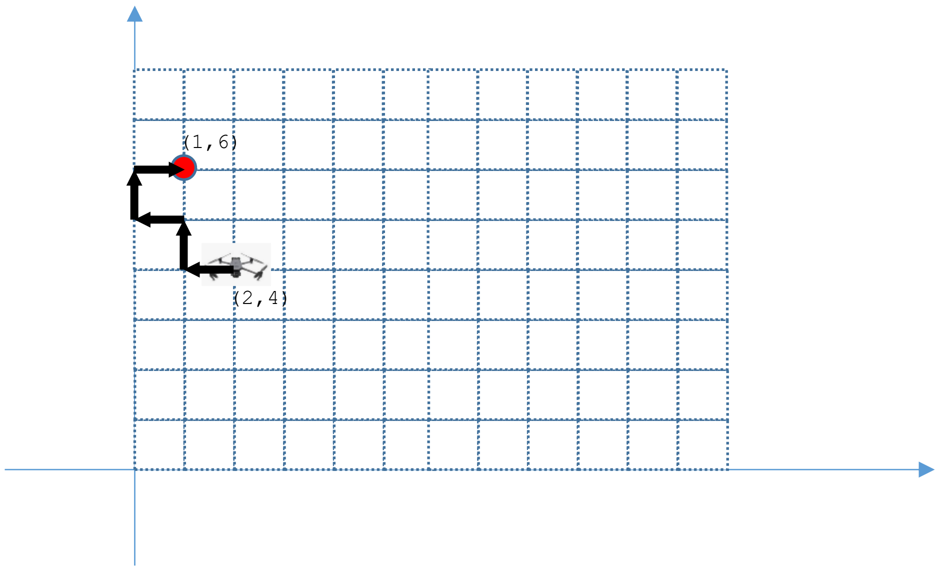
In this part (Part 1), there are some rules:

- In **Tasks 1 and 2**, you cannot use any built-in or import functions, including the member functions of built-in data type, e.g. `replace()`, `count()`, etc. Except, however, you can use the function `len()`.
- For each task, you can only write **one** function.

In this part, you are controlling a drone in a 2D plane. The drone starts at a 2D coordinates `(x,y)` where `x` and `y` are integers. The drone can move north, east, south or west using the commands '`N`', '`E`', '`S`' or '`W`' respectively. In short, if the drone moves:

- `'N' : y = y + 1`
- `'E' : x = x + 1`
- `'S' : y = y - 1`
- `'W' : x = x - 1`

For example, if the drone starts at position `(2,4)`, and it moves in the sequence west, north, west, north, and then east. It will end up at position `(1,6)`. The detail is shown in the picture below.



The sequence of move will be given in a string. For example, the above moves will be given as:

`'WNWNE'`

Task 1 Find the Final Position (Iterative Version)

Write an ***iterative*** `final_pos_i(x,y,seq)` to return the final position after a sequence of moves `seq` by the beginning position `(x, y)`. The input `seq` is a string that will not contain any character other than '`N`', '`E`', '`S`' or '`W`'. Your function must use for-loops or while-loops and cannot be recursive. Here are some sample output:

```
>>> print(final_pos_i(2,4,'WNWNE'))
(1, 6)
>>> print(final_pos_i(9,1,'NSWSSWNSSE'))
(8, -2)
>>> print(final_pos_i(0,0,'WWENSWSNSWSSWNSSE'))
(-3, -4)
```

Task 2 Find the Final Position (Recursive Version)

Write a ***recursive*** version `final_pos_r(x,y,seq)` with the same functionality of `final_pos_i` in the previous task. You cannot use any loops or list comprehension in this task.

Task 3 Find the Final Position (One-line Version)

In this part, you can use any built-in functions but not any imported functions.

Write a one-line function `final_pos_o(x,y,seq)` with the same functionality of `final_pos_i` in the previous task. By "one-line", your function body must have only one return line only like the below, without any semi-colon or calling other functions in Tasks 1 or 2.

```
def final_pos_o(x,y,seq):
    return #some value that is only one line
```

Part 2 DNA Mutations (15 + 5 + 10 = 30 marks)

In each task of this part, you are allowed to call the functions you wrote from the other tasks of this part.

DNA is a long string of amino acids. Usually there are only four types and they are represented by four alphabets, `'A'`, `'C'`, `'G'` and `'T'`. A normal DNA will be a string like:

`'GCAATGGAACCTACT...'`

In a space voyage, a researcher discovered an alien that can take in a single amino acid, and mutate it into a long gene as follows:

- `'A'` → `'CG'`
- `'C'` → `'AT'`
- `'G'` → `'CA'`
- `'T'` → `'GT'`

And it can repeat for many generations. For example, if the starting gene is 'A' and we want to mutate for 3 generations:

`'A'` → `'CG'` → `'ATCA'` → `'CGGTATCG'`

For the first generation, 'A' will be mutated to 'CG'. Then 'C' and 'G' will be mutated to 'AT' and 'CA' and become 'ATCA' in the second generation, and so on.

Task 1 Mutate

Write the function `mutate(amino_acid,no_gen)` to return a string of the DNA after `no_gen` generations from the amino acid `amino_acid`. You can assume the input `amino_acid` is always one single letter of `'A'`, `'C'`, `'G'` or `'T'` and `no_gen` ≥ 0. Here is some sample output:

```
>>> print(mutate('A',3))
CGGTATCG
>>> print(mutate('T',2))
CAGT
>>> print(mutate('G',5))
CGGTATCGATCGCAGTATCACAGTCGGTATCA
```

Task 2 Locating Amino Acid in the Mutated DNA

Write a function `amino_acid(start_amino_acid,no_gen,pos)` to return the amino acid at position `pos` in the mutated DNA after `no_gen` generations from the starting amino acid `start_amino_acid`. Note that the position pos observe the same rules as our string, e.g. the leftmost position is `pos=0`.You can assume

```
0 <= pos < 2**no_gen
```

Here are some example output:

```
>>> print(amino_acid('A',3,3)) # DNA:'CGGTATCG' index=3 but it is the fourth
T
>>> print(mutate('G',5))
CGGTATCGATCGCAGTATCACAGTCGGTATCA
>>> print(amino_acid('G',5,12))
C
```

Task 3 Locating Amino Acid in the Mutated DNA in Many Generations

You will gain points from this part if you can handle very big `no_gen` within 1 second.

```
>>> print(amino_acid('T',31,1231311))
A
```

Part 3 Burger Upgrade (30 marks)

Remember our burger exercises in our tutorials? We represent a burger by a string and each character in the string is one layer of the burger, e.g. `'B'`= a piece of the bun, `'C'` = cheese, `'P'` = beef patty and `'L'` = lettuce, etc. And an example burger will be something like `'BCLPCB'`.

In a given menu, now we offer any upgrade of our existing burgers! Given an initial burger, you can only upgrade that burger by adding more ingredients WITHOUT messing up the original order of ingredients.

For example, if the initial burger is `'BCPB'`, you can upgrade it to `'BLLCPLLB'` by adding more lettuce into it. So, the `'BLLCPLLB'` is an upgrade of the burger `'BCPB'`. However, the burger `'BPCLB'` is ***NOT*** an upgrade of the burger `'BCPB'` because the original order of `'BCPB'` is messed up. The same burger is also an "upgraded" version of itself.

Write a function `upgrade_burger(b1,b2)` to return `True` if burger `b1` is an upgraded burger of `b2`, otherwise, return `False`. The inputs `b1`and `b2` are strings that will not contain any character other than `'B'`, `'C'`, `'L'` or `'P'`.

Here are some sample output:

```
>>> print(upgrade_burger("BLLB","BLB"))
True
>>> print(upgrade_burger("BLB","BCLB"))
False
>>> print(upgrade_burger("BCLPCPLCB","BCLLCB"))
True
>>> print(upgrade_burger("BCLPLCLPPCLCPLCB","BCLPCPLCB"))
True
>>> print(upgrade_burger("BCLB","BLCB"))
False
>>> print(upgrade_burger("BCLCB","BLCB"))
True
>>> print(upgrade_burger("B"+"LC"*10000+"LB","B"+"LC"*10000+"CB"))
False
>>> print(upgrade_burger("B"+"LC"*10000+"VB","B"+"LC"*10000+"LCB"))
False
>>> print(upgrade_burger("B"+"LC"*10000+"CLB","B"+"LC"*10000+"CB"))
True
```

Code Template

```
#Part 1
def final_pos_i(x,y,seq):
    return #modify this

def final_pos_r(x,y,seq):
    return #modify this

def final_pos_o(x,y,seq):
    return #modify this

# You should not submit the following test cases
# print(final_pos_i(2,4,'WNWNE'))
# print(final_pos_i(9,1,'NSWSSWNSSE'))
# print(final_pos_i(0,0,'WWENSWSNSWSSWNSSE'))


#Part 2
def mutate(amino_acid,no_gen):
    return #modify this

def amino_acid(start_amino_acid,no_gen,pos):
    return #modify this

# You should not submit the following test cases
# print(mutate('A',3))
# print(mutate('T',2))
# print(mutate('G',5))
# print(amino_acid('A',3,3)) # DNA: 'CGGTATCG'
# print(mutate('G',5))
# print(amino_acid('G',5,12))
# print(amino_acid('T',31,1231311))



#Part 3
def upgrade_burger(b1,b2):
    return #modify this


# You should not submit the following test cases
# print(upgrade_burger("BLLB","BLB"))
# print(upgrade_burger("BLB","BCLB"))
# print(upgrade_burger("BCLPCPLCB","BCLLCB"))
# print(upgrade_burger("BCLPLCLPPCLCPLCB","BCLPCPLCB"))
# print(upgrade_burger("BCLB","BLCB"))
# print(upgrade_burger("BCLCB","BLCB"))
# print(upgrade_burger("B"+"LC"*10000+"LB","B"+"LC"*10000+"CB"))
# print(upgrade_burger("B"+"LC"*10000+"VB","B"+"LC"*10000+"LCB"))
# print(upgrade_burger("B"+"LC"*10000+"CLB","B"+"LC"*10000+"CB"))
```