PE1 (cs1010e, 2023-24 - sem1)

Programming Methodology (National University of Singapore)

CS1010E Practical Exam 1

Instructions:

- Your code will be graded and run on **Python 3.10**.
- 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 (except Part 1 Tasks 1 and 2). 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()`.
- 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.**
- Before the PE ends, remember to
    - Save your three parts in three files in your laptop
    - 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 Card Game (10+10+10 = 30 marks)

For Tasks 1 and 2, 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:

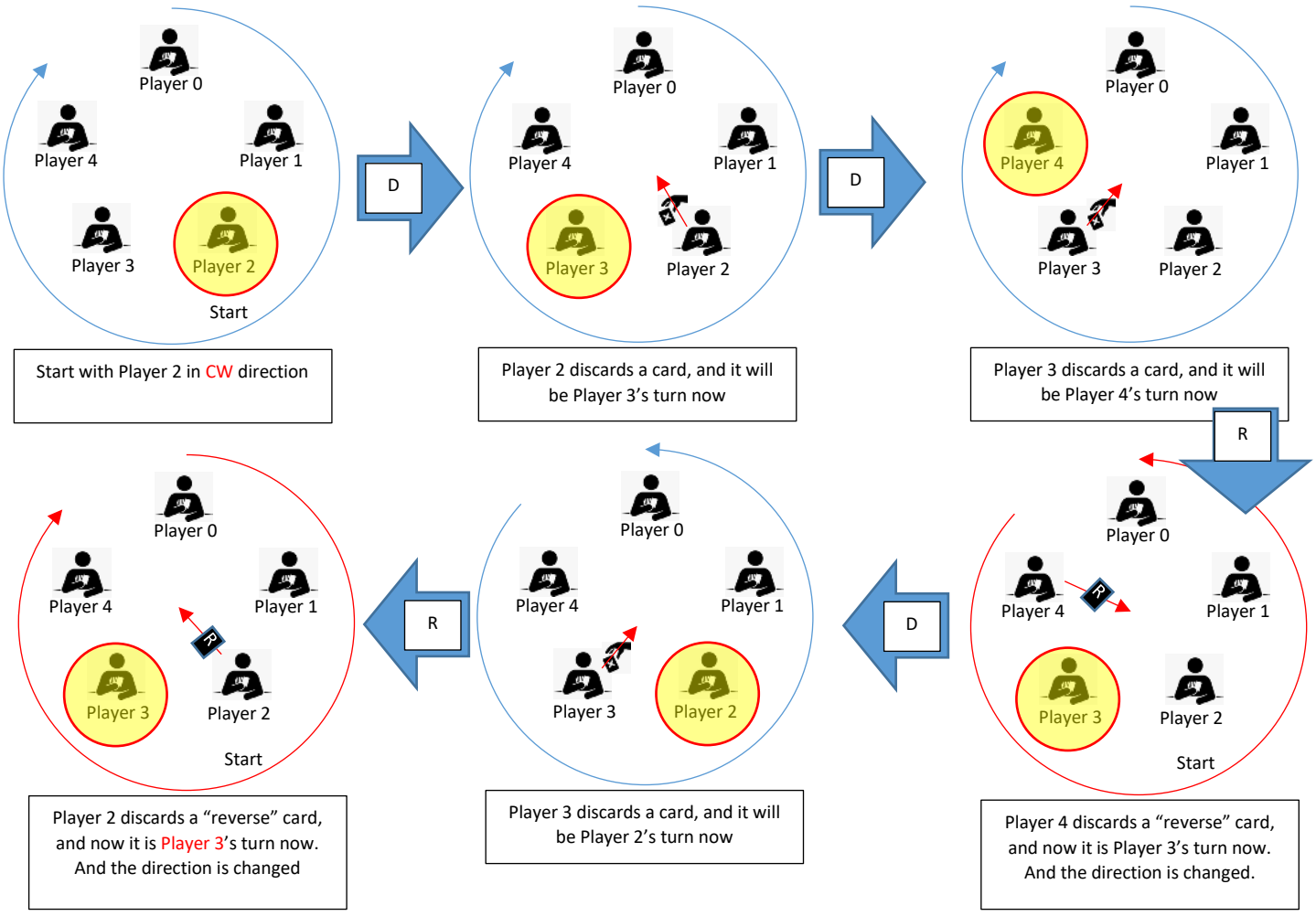
- `encode(), decode(), replace(), partition(), rpartition(), split(), rsplit(), translate(), map(), count(), find(), join(), rfind(), rsplit(), rstrip(), strip(), sort(), sorted(), pop()`

We are playing a card game similar to Uno. There will be N players in the game and we will label the players as Player 0, Player 1, Player 2, … , Player N-2 and Player N-1 for N > 1 and they are all sitting in a clockwise manner. To start the game, we can start with any player with *either* a clockwise or counter clockwise direction. If a player discard a non-functional card, then the next player is the player next in that direction. However, there is a functional card called "Reverse" that can reverse the direction of the game. Namely, if the current direction is clockwise, then the game will go counter clockwise, and vice versa. After the reverse, the next player will be the "next" one in the *new* direction. We will use an alphabet to represent each of the above actions:

- 'D' : Discarding a non-functional card
- 'R' : Reversing the direction

Here is a demonstration of a game play. Assume N = 5 and we start with Player 2 with starting clockwise direction. If our records of the players are

`D, D, R, D, R`



We will give the sequence of player actions in a string. For example, the above records will be

'DDRDR'

With this input, the final next player will be Player 3.

Task 1 Card Game Iterative Version (10 marks)

Write an ***iterative*** function `card_game_i(N,start_pos,start_dir,seq)` which **returns** the final next player number of the game after the actions are finished. Your return value `n` should be between `0` and `N-1`, namely, $0 \leq n \leq N-1$. Your function cannot be recursive in this task. The input `N` is the number of players in the game. The input `start_pos` is the number of the first player before any action is taken. The input `start_dir` is the starting direction of the game, either '`CW`' (clockwise) or '`CCW`' (counter clockwise). The variable `seq` is the sequence of actions mentioned above. Here are some examples:

```
>>> print(card_game_i(5,2,'CW','DDRDR'))
3
>>> print(card_game_i(7,6,'CCW','DDDDDRDDDRDDDDDDDDDDRRDDRDDDDD'))
5
>>> print(card_game_i(11,7,'CW','DDRDDDDRDRDDRDRRDRDDD'*900))
7
```

Task 2 Card Game Recursive Version (10 marks)

Write a ***recursive*** version `card_game_r(N,start_pos,start_dir,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, or call other function(s) other than `card_game_r()`.

Task 3 Add "Skip card" (10 marks)

Write a function `card_game_s(N,start_pos,start_dir,seq)` that can play the game above, plus one more function card called 'Skip'. If a skip card is discarded, then the game will skip the next player and pass the turn to the next next player. The action in the given sequence will be given as the alphabet '`S`'. You can choose whatever iterative or recursion version here that will give you the best performance. Here are some examples:

```
>>> print(card_game_s(6,4,'CW','DDRDSDDRDSDD'))
0
>>> print(card_game_s(101,57,'CCW','DDRDSRDSRSSDRDSRS'*999))
60
```

Unlike Tasks 1 and 2, you can write more helper function(s) to be called by `card_game_s()` in Task 3. You can even call the two functions in Tasks 1 and 2. Alternatively, you can just implement this feature into the functions in Task 1 or Task 2 and just call that in Task 3, and it will not deemed as redundant code.

Part 2 Find the Parents of a Monocell (20 + 15 = 35 marks)

Dr. Mad finished his crazy research that he invented a new species of single cells called the *Monocell*. The very special features of this type of cells are:

- Each of the cell has only one strand of DNA (which is a string of different combinations of the four letters 'A', 'C', T' and 'G'.) and it can be in any length more than zero
- Two cells can 'mate' with each other and produce a child Monocell such that the child's DNA is the concatenation of his parents.
- However, two cells with the same DNAs cannot mate with each other and produce a new child.

Task 1 Find the possible parents (20 marks)

You will be given a list of strings as a database of some DNAs of Monocells. Given a child DNA, write a function `find_parents(child_dna,dna_database)` to return a list of possible pairs of parents that can produce that child Monocell where each pair is expressed as a tuple. Note that the orders of the parents matter in each tuple. The child's DNA must be the one of the first parent followed by the second one in a tuple. For example, the second example below cannot be `[('G', 'ATGAT')]`. If you cannot find the parents in the database, return an empty list. Here are some examples:

```
>>> print(find_parents('ACGTA',['ACT','TA','CGTA','ACG','A','ACGT']))
[('A', 'CGTA'), ('ACG', 'TA'), ('ACGT', 'A')]

>>> print(find_parents('ATGATG',['ATGAT','ACT','GAT','ATG','G']))
[('ATGAT', 'G')]
```

Task 2 Find the possible parents from a huge database (15 marks)

Your `find_parents()` function in Task 1 must be able to handle very large lists within 1 second, i.e. lists of length > 1000000. For example, the following code will generate a list of more than 1000000 DNAs as the database for you to test your code. But do not submit the following code.

```
from itertools import product
w10 = [''.join(x) for x in product(list('ACGT'),repeat=10)]
```

And your function should be able to find the parents in the above database within 1 second:

```
>>> print(find_parents('ACGTTTTTTAATATTTATGG',w10))
[('ACGTTTTTTA', 'ATATTTATGG')]
```

You do not need to write another function for this task. Namely, your function in Task 1 should satisfy this criteria for Task 2.

Part 3 Three Little Pigs Defence (35 marks)

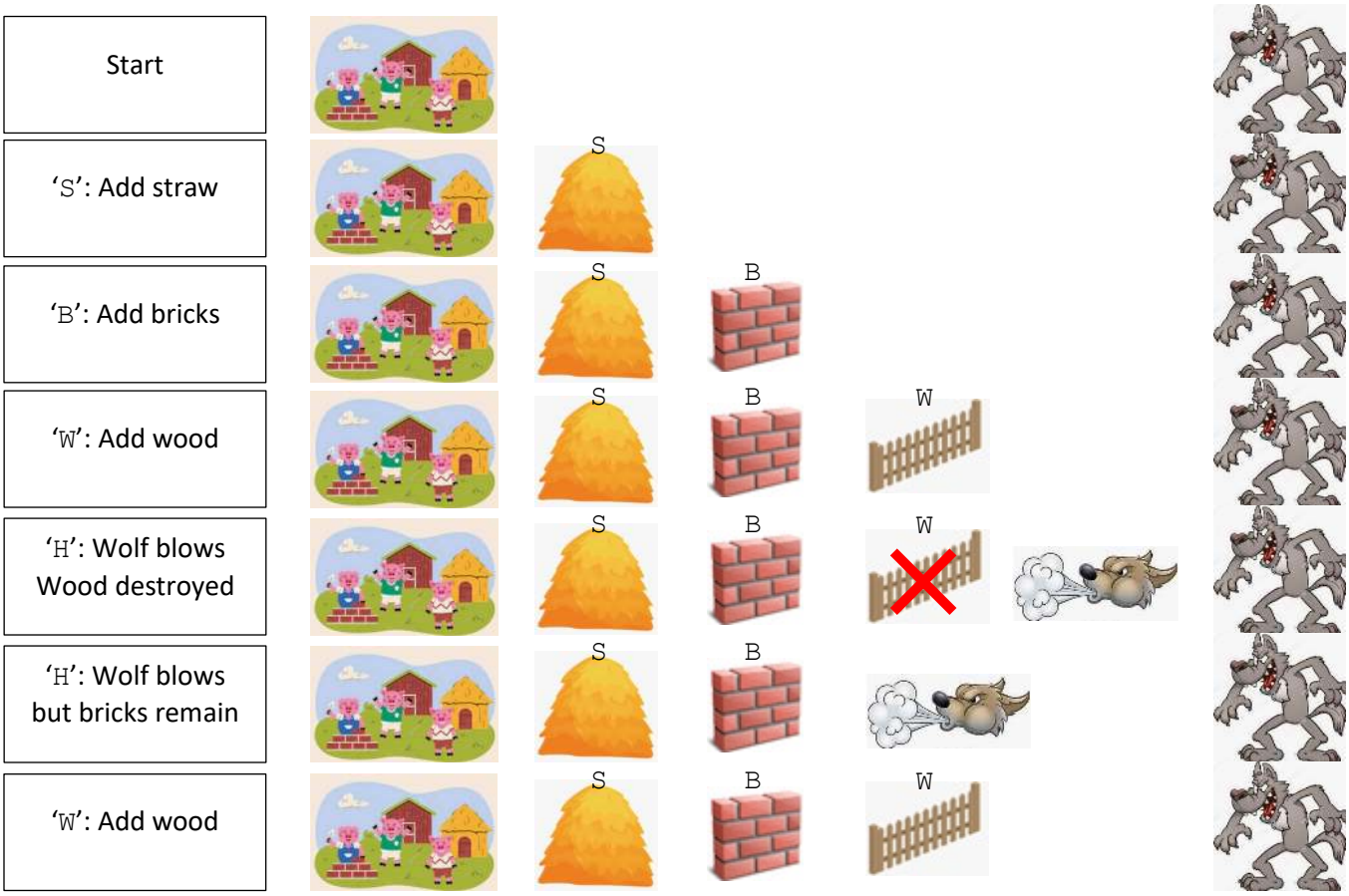
We heard of the story of the three little pigs. Here is the sequel when the Big Bad Wolf wants to come back to attack the three little pigs. Imagine that the three little pigs and the wolf are at the two ends of a road and they can only reach each other by that road. The pigs and the wolf will take some actions and there are four possibilities, represented by the four alphabets:

- `'S'` : The first little pig adds a **straw** barrier to stop the wolf
- `'W'` : The second little pig adds a **wooden** barrier to stop the wolf
- `'B'` : The third little pig adds a **brick** barrier to stop the wolf
- `'H'` : The wolf attacks and **huff-and-puff** (blows off) one layer of the pigs' barriers away

Each time a barrier is added, it is added at the "outer most" place that is nearest to the wolf. Each time the wolf attacks, he can blow off one layer of the barriers if the last barrier is made of straw or wood. However, if the last layer of the barriers is made of brick, the wolf's attack has no effect. One more thing, if there is no barrier left and the wolf came to blow. The wolf's attack is wasted because there is nothing to be blown away (and he doesn't want to blow away the pigs).

The events will come in a sequence given as a string. Your job is to compute what are the barriers left after all actions are taken. For example, if the sequence of event is:

- The first little pig adds a straw barrier
- The third little pig adds a brick barrier
- The second little pig adds a wooden barrier
- The wolf huff-and-puff
- The wolf huff-and-puff
- The second little pig adds a wooden barrier



The above events will be given in a string:

'SBWHHW'

And finally, the leftover is a barrier of straw, a barrier of brick and a barrier of wood that is represented by the following string as an output:

'SBW'

Task

Write a function `three_little_pigs_defence(seq)` to take in a sequence of actions `seq` and return the final string of what is left over after all events happened with `len(seq)` ≥ 0. Your return string should have the correct order according to the events. Here are some examples

```
>>> print(three_little_pigs_defence('SBWHHW'))
SBW
>>> print(three_little_pigs_defence('SHHWSHB'))
WB
>>> print(three_little_pigs_defence('BSHHSWHS'))
BSS
```

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 card_game_i(N,start_pos,start_dir,seq):
    return

def card_game_r(N,start_pos,start_dir,seq):
    return

def card_game_s(N,start_pos,start_dir,seq):
    return
```

Part 2

```
def find_parents(child_dna,dna_database):
    return

'''
# the following code is for testing only, do NOT submit them
from itertools import product
w10 = [''.join(x) for x in product(list('ACGT'),repeat=10)]
print(find_parents('ACGTTTTTTAATATTTATGG',w10))
'''
```

Part 3

```
def three_little_pigs_defence(seq):
    return
```

-- End of Paper --