National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester I, 2019/2020

Sidequest 7.1
Lazy Susan

Release date: 19 September 2019
Due: 30 September 2019, 23:59

Required Files

- sidequest07.1-template.py
- lazy_susan.py

Background

You happily sit in the train, satisfied that you managed to figure out the problem and get it running smoothly again. Strangely enough, the train seems to be running a little *too* smoothly. In fact, the train doesn't seem to have stopped in the last hour! You look around in shock as wisps of smoke slowly fill the cabin. A genie manifests through the smoky haze and presents you with a... Lazy Susan.

About the Lazy Susan Problem

Dating back to 1979, a logic puzzle also known as the blind bartender's problem was publicised by Martin Gardner in his "Mathematical Games" column. Four coins are placed at the corners of a square lazy susan in either heads or tails orientations. At every turn, the blindfolded player is required to pick and feel any two coins. The player then chooses whether to flip any of them after determining their orientation, after which the lazy susan will be rotated at a **random** angle. The original puzzle requires the blindfolded player to devise an algorithm that guarantees the same orientation for all coins within a finite number of turns.

Of course, the solution to the original problem can be found easily on the Internet, so the genie decides to use its own magical variant. The hazy mist swirls around your body and numbs your skin, robbing you of your ability to identify the coin orientations on the rotating table by touch. The genie silently indicates to you to start selecting the coins you wish to flip in this turn. Looks like you are the blind bartender tonight, and you will have to play to win if you ever want to get off the train...

This mission consists of **5** tasks.

Demonstration: Table ADT

The friendly neighbourhood genie has implemented a Table Abstract Data Type (ADT) representing the coins on the rotating table. Recall that with ADTs we do not need to know how data is stored — simply knowing the behaviour of the methods in the abstraction is sufficient. A disembodied voice echoes in the cabin, informing you of your very limited choice of allowable functions found in `lazy_susan.py` to manipulate said Table.

`create_table(size)` Returns a new Table with *size* number of coins.
`get_table_size(Table)` Returns the number of coins on the table.
`get_table_state(Table)` Returns a tuple representing the coin orientations on the table.
`check_solved(Table)` Returns `True` if all the coins are facing the same direction, otherwise returns `False`.
`flip_coins(Table, move=None)` Has no return value. Flips coins on the table at specified indices in the boolean *move* tuple, with length corresponding to the number of coins on the table. If *move* is not specified, coins on the table are randomly chosen to be flipped. The `Table` is then spun randomly.
`run(Table, Solver)` Optional. Has no return value. Runs the user-implemented Solver algorithm (e.g. solve_4, solve), printing the state of the table at each turn.

The Table ADT provided works similarly to the blind bartender's problem — during a single turn, you specify whether the coin at each index of the *move* tuple should be flipped, after which the Table will be randomly (the genie promises) rotated before the next turn commences.

Here are some examples that you might find useful:

```
# Creating a Table object with 4 coins
my_table = create_table(4)

# Asking the genie to tell you the number of coins on the table
get_table_size(my_table)
#=> 4

# Begging the genie to tell you what is the current state of the table
get_table_state(my_table)
#=> (1,0,1,0)

# Asking the genie if you have completed the problem
check_solved(my_table)
#=> False
```

```
# Flipping the first and last coins with the move (1,0,0,1)
# and rotating the table
flip_coins(my_table, (1,0,0,1))

get_table_state(my_table)
#=> (0,1,1,0)
# rotated from (0,0,1,1)

# Asking the genie if you have completed the problem
check_solved(my_table)
#=> False

get_table_state(my_table)
#=> (0,1,1,0)

# Flipping the middle two coins with the move (0,1,1,0) and rotating
flip_coins(my_table, (0,1,1,0))

get_table_state(my_table)
#=> (0,0,0,0)
# rotated from (0,0,0,0)

# Random flipping
flip_coins(my_table)

get_table_state(my_table)
#=> (1,1,0,1)
# rotated from (1,0,1,1)

# Flipping the first two and last coins and rotating
flip_coins(my_table, (1,1,0,1))

# Asking the genie if you have completed the problem
check_solved(my_table)
#=> True
```

As an added visualization aid, we have provided you with an interactive GUI function `Susan` also found in `lazy_susan.py` to experiment with your algorithms. The GUI function takes in a Table as argument.

- Click on the individual dots to select which coins to flip.
- To flip selected coins and rotate the table, press the 'Space' key.
- To show the current state of the table, press the 'D' key.

```
# Creating a Table object with 4 coins
my_table_2 = create_table(4)

# Invoke the Susan GUI spell with the table of 4 coins
Susan(my_table_2)
```

Task 1: Two Coins (2 marks)

The genie sets a table of 2 coins in front of you as a warm-up. The haze smells a little funky, and reminds you faintly of burnt forest, but it doesn't stop you from taking blatant peeks at the table, which makes the problem pretty trivial.

Given a table of 2 randomly-oriented coins, your task is to define a function **`solve_trivial_2`** that takes in a `table` as input and solves it in one turn. Using the function `get_table_state`, identify the correct move to use to solve the table. If you are having trouble interfacing with the Table ADT, use the `run` function to visualize your algorithm.

```
t2_1 = create_table(2) # Creating a Table object with 2 coins
run(t2_1, solve_trivial_2)
```

Task 2: Four Coins (2 marks)

Seeing your expert hand, the genie replaces the table with 4 coins. Still, you realise that unless you are blindfolded, nothing can stop you from peeking at the table. Again, the problem is trivial.

Given a table of 4 randomly-oriented coins, your task is to define a function **`solve_trivial_4`** that takes in a `table` as input and solves it in one turn. Using the function `get_table_state`, identify the correct move to use to solve the table. If you are having trouble interfacing with the Table ADT, use the `run` function to visualize your algorithm.

```
t4_2 = create_table(4) # Creating a Table object with 4 coins
run(t4_2, solve_trivial_4)
```

Task 3: Two Coins Blind (4 marks)

The genie figures out that you are cheating by peeking at the `table` and is unhappy. Another table of 2 coins is set in front of you as the haze intensifies and thickens, effectively removing your sense of sight. It gradually dawns on you that the challenge lies in the lack of information on the state of the table at any point of time, even more so when the table is continuously rotated after every turn — you'll have to trust the genie to be honest when the table is finally solved.

Given a table of 2 randomly facing coins, your task is to define a function **`solve_2`** that takes in a table as input and solves it with the algorithm described below. You are **not** allowed to use `get_table_state` any longer. Random flipping is a great way to kill time, but the genie is confident that you will be able to come up with a deterministic solution.

```
t2_3 = create_table(2)
run(t2_3, solve_2)
```

The N=2 algorithm

Begin by checking if the table is already solved. If it is then we are done. Otherwise, it must mean that the table has 1 heads and 1 tails, so just flip any of the two coins and we must have both facing the same side up.

Task 4: Four Coins Blind (4 marks)

Happy that you are now playing the game correctly, the genie replaces the table of 2 coins with a table of 4 coins.

Given a table of 4 randomly facing coins, your task is to define a function **`solve_4`** that takes in a table as input and solves it with the algorithm described below. Again, you are **not** allowed to use `get_table_state`. Random flipping is a great way to kill time, but you'll need something more reliable if the genie ever decides to throw at you a more complex table.

```
t4_4 = create_table(4)
run(t4_4, solve_4)
```

The N=4 algorithm

Figure 1 shows how a sample execution of the algorithm will look like, with rotations representing a right shift. Note that the table can reach a solved state at any point in time (guaranteed within 7 ($2^3 - 1$) moves). The procedure is given by:

```
A B A C A B A
```

	1	1	1	1
A	1	0	1	0
B	1	1	0	0
C	1	0	0	0

Table 1: Set of moves for $N = 4$

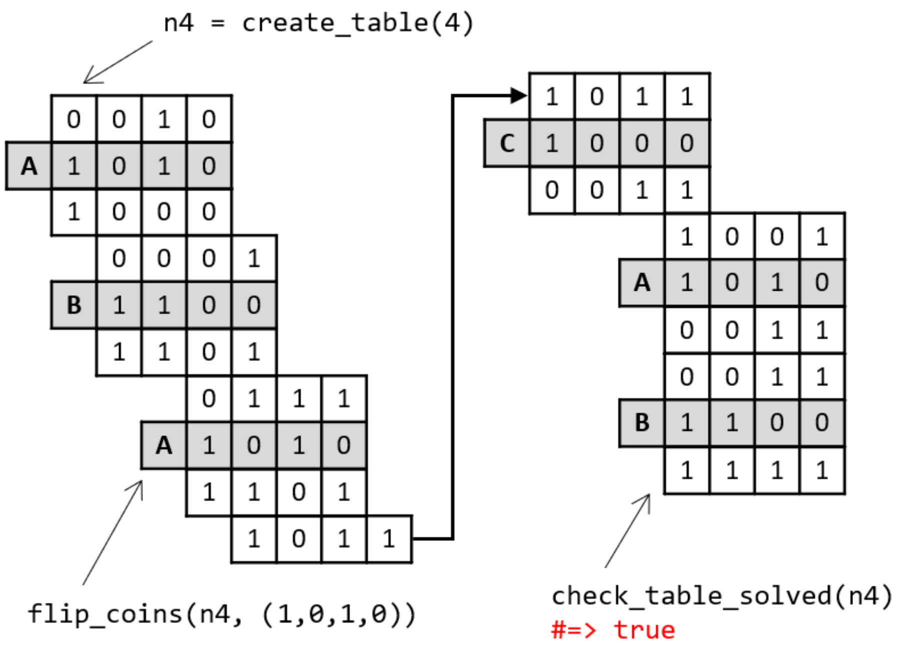

Figure 1: Step-by-step execution for $N = 4$

Task 5: The Mystery of 2^n Coins (2 marks)

The genie snickers. This doesn't *look* good. You sense a table placed before you, a much heavier one, but gosh — you have no idea how many coins there are. Calling upon the inherited wisdom of foregone PIM wizards before you, you discover that only tables with 2^n coins can be solved deterministically.

The genie concurs.
Can you figure out why? Post on the forums to share your thoughts.

Meanwhile, distracted as you may, you need to think of a solution for 2^n coins. Fast. Your dinner at home is in jeopardy while the haze proceeds to begin numbing your sense of taste. Given a table of *N* randomly facing coins, where *N* is a power of 2, your task is to define a function `solve` that takes in a table as input and solves it. Of course, you are **not** allowed to use `get_table_state`.

The general algorithm

It will be helpful to do a little pattern searching using the case for $N = 8$, and comparing it with $N = 4$. For $N = 8$, the solution is guaranteed in 127 ($2^7 - 1$) turns using the procedure given by:

```
*D*E*D* F *D*E*D* G *D*E*D* F *D*E*D*
```

	1	1	1	1	**1**	**1**	**1**	**1**
A	1	0	1	0	**1**	**0**	**1**	**0**
B	1	1	0	0	**1**	**1**	**0**	**0**
C	1	0	0	0	**1**	**0**	**0**	**0**
D	**1**	**1**	**1**	**1**	0	0	0	0
E	**1**	**0**	**1**	**0**	0	0	0	0
F	**1**	**1**	**0**	**0**	0	0	0	0
G	**1**	**0**	**0**	**0**	0	0	0	0

Table 2: Set of moves for $N = 8$

where `*` = `ABACABA` from the case of $N = 4$. Notice how the procedure is similar to the iterative solution for Tower of Hanoi (or its corresponding rotating backup scheme), while the corresponding set of moves is iteratively built from $N = 4$.

Disk 1	A		A		A		A		A		A		A		A		A		
Disk 2		B				B				B				B				B	
Disk 3				C								C							...
Disk 4								D											
Disk 5																E			
⋮									⋮										⋱

Figure 2: Tower of Hanoi backup rotation scheme

Use these hints to write the general function `solve` to solve for all 2^n coins, $n \geq 1$ ($N = 2, 4, 8, 16, \ldots$). Kudos if $n \geq 0$ ($N = 1$) can also be solved with the same solver.

A word of warning though: It is not advisable to execute `run()` for large table sizes, which can print an excessive number of lines. Proceed at your own risk!

Conclusion

Probably satisfied, the genie vanishes with a poof along with the funky haze. Congratulations! You find your dinner less palatable than usual, but the taste is still manageably bittersweet. Hopefully, the genie won't target your supper later...