CSCI 1133, Fall 2019
Programming Assignment 7
Due: 11:55pm, Wednesday October 23, 2019

Due Date: Submit your solutions to GitHub by 11:55 p.m., Wednesday, October 23rd. We will do a pull from this time point. Do not upload anything to Canvas and PLEASE be sure to use proper naming conventions for the file, classes, and functions. We will NOT change anything to run it using our scripts.

Unlike the computer lab exercises, this is not a collaborative assignment. You must design, implement, and test your code on your own without the assistance of anyone other than the course instructor or TAs. In addition, you may not include solutions or portions of solutions obtained from any source other than those provided in class (so you are ONLY allowed to reuse examples from the textbook, lectures, or code you and your partner write to solve lab problems). Otherwise obtaining or providing solutions to any homework problem for this class is considered Academic Misconduct. See the syllabus and read section "Academic Dishonesty" for information concerning cheating. Always feel free to ask the instructor or the TAs if you are unsure of something. They will be more than glad to answer any questions that you have. We want you to be successful and learn so give us the chance to help you.

Instructions: This assignment consists of 3 problems, worth a total of 40 points. Solve the problems below by yourself, and put all functions in a single file called `hw7.py`. Use the signatures given for each class and function. We will be calling your functions with our test cases so you must use the information provided. If you have questions, ask!

Because homework files are submitted and tested electronically, the following are very important:

- You follow all naming conventions mentioned in this homework description.
- You submit the correct file, hw7.py, through Github by the due date deadline.
- You follow the example input and output formats shown.
- Regardless of how or where you develop your solutions, your programs should execute using the python3 command on CSELabs computers running the Linux operating system.

Push your work into Github under your own repo. The specific hosting directory should be: repo-<username>/hw7, where you replace <username> with your U of M user name. For instance, if your email address is bondx007@umn.edu, you should push your hw7 to this directory: repo-bondx007/hw7

The following will result in a score reduction equal to a percentage of the total possible points:

- Incorrectly named/submitted source file, functions, or classes (20%)
- Constraints not followed (40%)
- Failure to execute due to syntax errors (30%)
- Not filling out the documentation template for every function, other bad code style (10%)

Use the following template for EVERY function that you write (note: a helper function is a function so it gets a template.)

```
#==========================================
# Purpose: (What does the function do?)
# Input Parameter(s): (Each parameter by name and what it represents)
# Return Value(s): (What gets returned? Possibilities?)
#==========================================
```

Problem A. (10 *points*) **Collatz**

The Collatz conjecture (https://en.wikipedia.org/wiki/Collatz_conjecture) is an unproven mathematical rule that says the following:

Take any positive integer *n*. If *n* is even, divide it by 2 to get *n / 2* (use integer division so that you don't end up with floating point numbers). If *n* is odd, multiply it by 3 and add 1 to obtain *3n + 1*. Repeat the process indefinitely, and eventually you will reach 1.

Write a **recursive** function called `collatz` that takes in a single positive integer argument `n` and returns the list of numbers in the collatz sequence from `n` to 1, inclusive. For example, suppose your initial number was `5`. Then the collatz sequence would be the following:

The initial number is **5**.
5 is odd, so the next number is 5*3 + 1 = **16**
16 is even, so the next number is 16//2 = **8**
8 is even, so the next number is 8//2 = **4**
4 is even, so the next number is 4//2 = **2**
2 is even, so the next number is 2//2 = **1**
We have reached 1, so the sequence is complete, and the function would return the list `[5,16,8,4,2,1]`

Hints:

- This function returns a list, so both your base case and your recursive case(s) need to return a list.
- You will need to add elements to the **front** of the list in this problem.
- `.append` doesn't add things to the front of a list, so put the element you want to add in brackets to turn it into a list and then use list addition. To put element `x` at the front of the list `ls`, you'd use `[x] + ls`. For example, note that:
  `collatz(5) = [5]+collatz(16) = [5]+[16,8,4,2,1] = [5,16,8,4,2,1]`

Constraints:

- The collatz function must be implemented using only recursion. Do not use any loop constructs (while or for).
- Do not import any modules (except random, for problem C).
- You may assume that your function will only be called with positive integers.
- Don't use the input() function, as this will break our grading scripts.

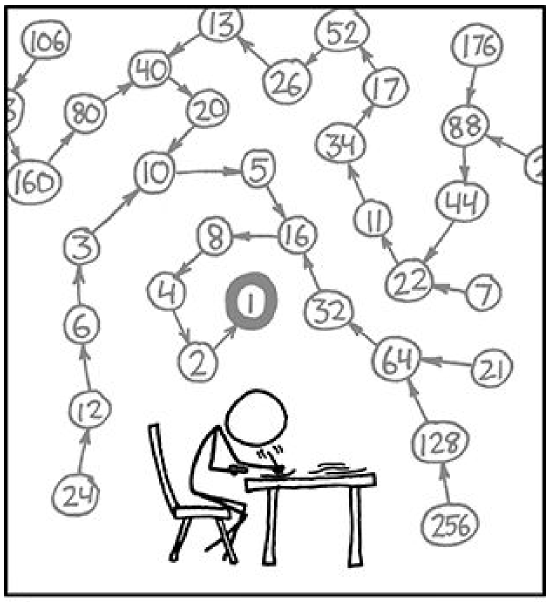
- Your submission should have no code outside of function definitions, except for comments and import statements.

Examples:

```
>>> collatz(5)
[5, 16, 8, 4, 2, 1]

>>> collatz(1)
[1]

>>> collatz(123)
[123, 370, 185, 556, 278, 139, 418, 209, 628, 314, 157, 472, 236,
118, 59, 178, 89, 268, 134, 67, 202, 101, 304, 152, 76, 38, 19, 58,
29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2,
1]
```



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

Image source: https://xkcd.com/710/

Problem B. (10 *points*) **Finding the Minimum**

Write a **recursive** function `find_min` that takes in one argument, a list of integers `num_list`, and returns the minimum value in that list. You may assume that `num_list` contains at least one element. Note that even if there is a tie, there should still only be one minimum value (it just appears more than once in the list).

Constraints:

- The `find_min` function must be implemented using only recursion. Do not use any loop constructs (while or for).
- Built-in functions like `min` and `sorted` that trivialize the problem are banned, for obvious reasons.
- Do not import any modules (except random, for problem C).
- Don't use the input() function, as this will break our grading scripts.
- Your submission should have no code outside of function definitions, except for comments and import statements.

Examples:

```
>>> find_min([8])
8

>>> find_min([0, 2, -5, -2, 5, -1, 4, 0, -5, -1])
-5

>>> find_min([20, 34, 32, 34, 48, 43, 21])
20
```

Problem C. (20 *points*) **Playing Tic-Tac-Toe Perfectly**

In Homework 5, we created a program to play tic-tac-toe by choosing an empty spot at random. This is generally a very bad strategy: most preschoolers can beat it more than half the time (even when they don't cheat). But, with the power of recursion, we can make a much better tic-tac-toe AI: in fact, we can make one that never loses. To do this, our AI will look at every possible sequence of moves and try to find a strategy that forces a win, or at least forces a draw if that's not possible.

Creating a tic-tac-toe AI that can't lose will require a few steps. You'll want to start by copying in all of your tic-tac-toe code from Homework 5: most of the functions there will be useful for this problem as well. We will post a solution to Homework 5B for those of you who didn't get all of the functions working.

Write a **recursive** function called `force_win(board)`, which takes in one argument: `board` is a list representation of a tic-tac-toe board (see Homework 5). `force_win` should return an integer that represents the current state of the board. Board state can be one of three values:

- **`1`** means that X has won, or can force a win: that is, there are moves that X can make so that no matter what O does, X will win
- **`-1`** means that O has won, or can force a win: that is, there are moves that O can make so that no matter what X does, O will win
- **`0`** means that neither of the above is true: if both players play perfectly, the game will end in a draw

This means that when looking at potential next moves, X will attempt to find moves that leave the board in the highest value for board state possible (because X would prefer to win (state 1), over a draw (state 0), over losing to O (state -1)).

Whereas O will attempt to find moves that leave the board in as low of a board state as possible (because O would prefer to win (state -1), over a draw (state 0), over losing to X (state 1)).

The general approach is as follows:

1. Figure out whose turn it is using the open_slots function. Since we start with 9 open spots and alternate turns, if there are an odd number of open slots it's X's turn, otherwise it's O's.
2. As your base cases, use the winner function and determine whether the game is already over. If it is, return the appropriate board state (1 for X winning, 0 for a draw, -1 for O winning).
3. If it is currently X's turn, then we want to find the move that would result in the highest possible value for the board state. So loop through the possible moves (that is, loop through the open slots in the board), and keep track of the board state you get back for each move: we will return the highest one. In particular, do the following for each possible move:
    a. Create a copy of the current board (you can use board[:] for this).
    b. Place the potential move on the copied board (put an X in that slot).
    c. Recursively call `force_win` on that board to see what the board state would be (think about why this is guaranteed to move towards a base case).
4. At the end of the loop, return the highest board state value that you found.

5. If it is O's turn, do the opposite: return the lowest possible board state you could get from any of O's potential moves.

Test this function using the examples below: only go on to the next part after you are sure that `force_win` works.

Finally, you will alter the `tic_tac_toe` function from Homework 5 to use your AI to decide moves for player O (X should still move randomly).

Each time O moves, you should loop through the possible moves (using the open_slots function), and call `force_win` on each of them, then choose the move that resulted in the lowest board state. Note that this is a little more complex than the loop within `force_win`, because you have to keep track of not only the lowest board state you have found, but also what move (i.e. what index in the board) got you that result.

Use the `play_games` function (also from Homework 5) to test the win rate of your AI. O should never lose.

Hints:

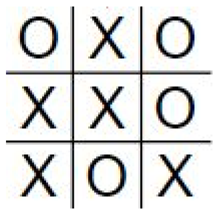
- Unlike the previous problem, you will need both a loop AND recursion in this function.

Constraints:

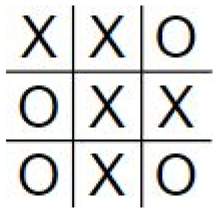
- `force_win` must use recursion (in addition to a loop).
- Do not import any modules except random.
- You may assume that your function will only be called with valid board states (you won't be given boards where both X and O have won).
- Don't use the input() function, as this will break our grading scripts.
- Your submission should have no code outside of function definitions, except for comments and import statements.
- Calling `force_win` on an empty board will probably take some time, since there are something like 300,000 possible board states to analyze, but it should still run in under a minute on any lab machine.

Examples (shown below each example is the board it represents, with the optimal move highlighted, if applicable):

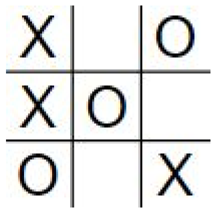
```
>>> force_win(['O', 'X', 'O', 'X', 'X', 'O', 'X', 'O', 'X'])
0
```



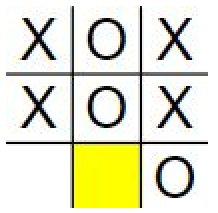
```
>>> force_win(['X', 'X', 'O', 'O', 'X', 'X', 'O', 'X', 'O'])
1
```



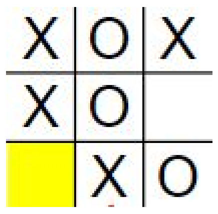
```
>>> force_win(['X', '-', 'O', 'X', 'O', '-', 'O', '-', 'X'])
-1
```



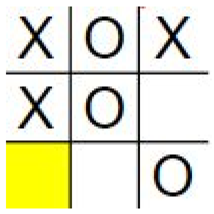
```
>>> force_win(['X', 'O', 'X', 'X', 'O', 'X', '-', '-', 'O'])
-1
```



```
>>> force_win(['X', 'O', 'X', 'X', 'O', '-', '-', 'X', 'O'])
0
```

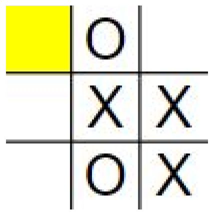


```
>>> force_win(['X', 'O', 'X', 'X', 'O', '-', '-', '-', 'O'])
1
```

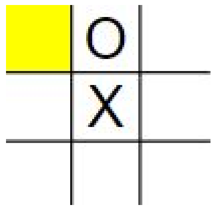


(Note: with the next three tests, the upper left corner is only one of multiple optimal moves in each case)

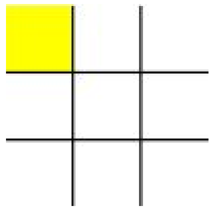
```
>>> force_win(['-', 'O', '-', '-', 'X', 'X', '-', 'O', 'X'])
1
```



```
>>> force_win(['-', 'O', '-', '-', 'X', '-', '-', '-', '-'])
1
```



```
>>> force_win(['-', '-', '-', '-', '-', '-', '-', '-', '-'])
0
```



(Note: This may take a while: 100 games can take several minutes to complete. Your number of draws vs. O wins will vary, but you should have zero X wins)

```
>>> play_games(1)
X wins: 0
O wins: 1
Draws: 0
```

```
>>> play_games(100)
X wins: 0
O wins: 73
Draws: 27
```