Function Name: `gcdLCM`

Inputs:

1. *(double)* A positive integer, *a*
2. *(double)* Another positive integer, *b*

Outputs:

1. *(double)* The GCD of *a* and *b*
2. *(double)* The LCM of *a* and *b*

Banned Functions:

```
gcd, lcm
```

Function Description:

Calculating the Greatest Common Divisor (GCD), and the closely related Least Common Multiple (LCM) of two numbers is a very common step in many cryptographic algorithms. For this problem you will be computing both!

One way to compute the GCD is to multiply common prime factors of *a* and *b*. However, finding the prime factorization of a number is a very expensive computation for your computer, which is why there is a faster algorithm called The Euclidean Algorithm. It relies on the fact that $gcd(a,b) = gcd(a+kb,\ b)$ where *k* is any integer. Basically, this property says that you can add or subtract a multiple of *b* from *a* and the GCD remains the same. Here's how the algorithm uses this property to calculate the GCD.

The Euclidean Algorithm:

1. While $a,b > 0$, find *r*, the remainder of a/b
2. Set $a = b$, $b = r$
3. Repeat
4. When $b = 0$, the GCD is a

Once the GCD has been found, the LCM of *a* and *b* can be found using the simple relation

$$lcm(a,b) = \frac{ab}{gcd(a,b)}$$

Function Name: `gorillaCase`

Inputs:

1. *(char)* A string of any length

Outputs:

1. *(char)* The modified input string, now in gorilla case

Banned Functions:

`length(), numel(), size(), ndims()`

Function Description:

After watching the new Planet Earth II trailer, you realize nature is amazing and decide to do whatever you can to protect it. It comes to your attention that the Western Lowland Gorilla is currently critically endangered due to an Ebola virus outbreak in 2003 and a poaching incident on May 28th, 2016. For these reasons, you choose to raise awareness for this majestic species by creating a special case dedicated to these animals.

To convert a string to gorilla case, you should first remove all non-letter characters and then capitalize the letters in odd positions of words which contain an odd number of letters. All other letters in the string should be lowercase. To clarify, all words of length 1, 3, 5, 7, etc. should have every other letter capitalized. For example, the string

`'Hello world, my name is JOHN CENA! Babadadaa!'`

would become

`'HeLlO WoRlD my name is john cena BaBaDaDaA'`

Notes:

- There will always be at least one space before each word, except the first word, which may have none.
- The total number of spaces should remain constant between the input and output string.

Hints:

- Try running the function with the name of one of the Western Lowland Gorillas from the nation's largest gorilla collection (think local).

Function Name: `checkers`

Inputs:

1. *(char)* An MxN array representing a checkerboard

Outputs:

1. *(double)* The number of possible jumps

Function Description:

Since we love games so much, let’s analyze a checkerboard! Assume you are the player with red pieces, and it’s your turn. Given a checkerboard where the capital letter ‘O’ represents empty spaces, ‘R’ represents red kings, ‘r’ represents red regular pieces, and ‘b’ represents any black piece, calculate the number of single jumps that can be made by moving any movable red pieces. In other words, how many possible jumps could you choose from on the next move?

All pieces jump forward (up) diagonally. Kings can also jump backwards diagonally. If there is a black piece immediately across the red piece on the diagonal, and the other side is an empty space, a jump can take place. For example, if a king is able to jump forward-left, forward-right, backwards-left and backwards-right diagonally over four different black pieces, that is counted as 4 possible jumps for just that one king. If you are still confused, here are some simple rules.

Notes:

- You can not jump off the board.
- Forward is always “up”.
- The checkerboard doesn’t necessarily have to be a realistic size or have a realistic placement of pieces.
- Even if it is possible to jump again after the first jump (a double jump, triple jump, etc.), you should only count the first jump.

Hints:

- For kings, there are four diagonals to check. For regular pieces, there are two.
- Try to represent the edges of the board in a way that makes this problem easier to solve.

Function Name: `pkmnBattle`

Inputs:

1. *(double)* A 1x7 vector representing the statistics of your Pokemon
2. *(double)* A 1x7 vector representing the statistics of your rival's Pokemon
3. *(double)* A 4x3 array representing the moves that your Pokemon knows
4. *(double)* A 4x3 array representing the moves that your rival's Pokemon knows

Outputs:

1. *(char)* A string describing the outcome of the battle

Background:

You are a Pokemon Trainer on an epic journey through the Midtown region. After collecting eight gym badges, defeating Team u(sic)ga, and beating the Elite Forty-Two CS 1371 TAs, you arrive at the Champion's Room and find your rival, George P. Burdell, who you must defeat to become the Pokemon Champion. In order to win this epic showdown, you think it will be helpful to know if your Pokemon or your rival's Pokemon will faint first, so you decide to code a function in MATLAB to help determine who wins!

Function Description:

The function takes in the statistics of both Pokemon as row vectors containing: `[Level, HP, Attack, Defense, Special Attack, Special Defense, Speed]`, always in that order. The function also takes in the four moves that each Pokemon knows. A Pokemon's moveset is depicted as a 4x3 array, where each row corresponds to a move and contains the `[Base Power, PP, Type]` of the move. PP is the number of times a move can be used, and Type is either 0, for physical moves, or 1, for special moves.

The amount of damage done by a move can be calculated from the formula below:

$$Damage = \left(\frac{2 \times Level + 10}{250} \times \frac{Attack}{Defense} \times Base + 2\right)$$

Base is the Base Power of the move and Level comes from the Pokemon using the move. If the move is physical, use the attacking Pokemon's Attack and its opponent's Defense, but if the move is special, use Special Attack and Special Defense.

Whichever Pokemon has the higher speed stat goes first. The Pokemon take turns attacking each other with its most powerful move, decreasing the other's HP by the amount of damage done by that move. Every time a move is used, its PP should decrease by one until it hits zero, at which point it can no longer be used. The first Pokemon that has 0 or fewer HP faints and loses. If you win the battle, the function should output the following statement:

```
'Congratulations, Champion of the Pokemon League! Your Pokemon survived
                 with <Your Pokemon's remaining HP> HP.'
```

If you lose, however, the function should output the following statement:

```
'You lost the battle and blacked out! The enemy had <Remaining enemy HP> HP
                              remaining.'
```

Notes:

- The speeds of the two Pokemon are guaranteed to be different.
- If the most powerful move is out of PP, use the second most powerful. If that is out of PP, use the third most powerful, etc.
- Both Pokemon are guaranteed to have enough PP to finish the battle.
- All input values will be non-negative.
- Round damage calculations to the nearest integer, before subtracting the damage from the opposing Pokemon.

Hints:

- Appropriate music for this problem

Extra Credit

Function Name: `conway`

Inputs:

1. *(logical)* An NxN start state for the cells in the Game of Life

Outputs:

1. *(logical)* The final state in the Game of Life

Function Description:

Conway's Game of Life is a simulation of cell reproduction and population. It is an automatic process, meaning that it is completely derived from an input state and a set of rules that will drive the simulation forward. There are a lot of cool patterns that the game can produce, and not all of them end, but some of them over time will reach a point where it settles and oscillates between two states forever. For this homework problem, your job is to perform Conway's Game of Life on an input array and give the resting states of the game.

The input to the function is a logical array where trues represent living cells in the system. At every tick of the game, the 'board' is updated by the following rules:

1. Any live cell with fewer than two live neighbors dies, as by underpopulation
2. Any live cell with two or three live neighbors lives on to the next generation
3. Any live cell with more than three live neighbors dies, as by overpopulation
4. Any dead cell with exactly three live neighbors comes back to life, as by reproduction

After an undefined time of iterating and updating the game boards, the boards will eventually oscillate between two final states. The output array is the combination of the two final states--a logical combination of cells in each of the two final states. That is, if there is a live cell in either of the two final states at an index, it should appear at the index in the output array.

Notes:

- The board will be square.
- As stated earlier, not every possible input state will end in an two oscillating states, but you can assume that any input that we give will end in two oscillating states.
- The board wraps around--for example, any cell in the first column has a neighbor on its left that is the same row in the last column. The same goes for the top row/bottom row and any combination of the two.
- The cells are 8 connected, which means that neighbors are not only horizontally/vertically adjacent but also diagonally adjacent.
- Every cell on the board should be updated 'simultaneously' such that in the middle of one update changing one cell will not affect how any of the other cells update.
- https://en.wikipedia.org/wiki/Conway's_Game_of_Life

Hints:

- Think of how logical operators & and | work on logical arrays to make your output.
- This can be a little hard to debug--to help you, we have provided a function to visualize what is going on. It is called `plotConway()` and takes in one input. The input is the board, and the function shows what the current board looks like in a plot. If you want to see what is going on, you can call the `plotConway()` function each time you update the board. NOTE: 1) Plotting slows the process down considerably, especially if you running it with a large board. We will not use large boards for the test cases but if you are making your own test cases keep this in mind. 2) Please **DO NOT KEEP** the `plotConway()` call in the final function that you submit--it will make the autograder slow down. Only use it as a debugging tool.
- If you want to see what the solution file looks like, there is a second optional input for `conway_soln()`. If you call the solution file with the string `'plot'` as the second input, it will plot what the solution file is running (i.e. if you just want to run the solution file with a board it is `conway_soln(board)`; if you want to see what is happening as well, it is `conway_soln(board, 'plot')`.)
- Continuing off of the last bullet, if you are having trouble wrapping your head around what should be happening (or want to see some of the cool patterns/animations that can be produced), try looking up Conway's Game of Life animations or run the MATLAB built in function `life()`, which launches a GUI that will run an animation of Conway's Game of Life with a random start.
- Use helper functions!