Drill Problem #1

Function Name: countNum

Inputs:

1. (*double*) An MxN array
2. (*double*) A particular number to search for

Outputs:

1. (*double*) A count of how many times the given number occurs in the array

Function Description:

Write a function in MATLAB named `countNum()` that will take in an input array of size MxN and compare each index within the array to find if it is equal to the second input. The output value should be equal to the number of values in the array that are equal to the second input.

Notes:

- The first input will always be a rectangular array of numbers. The second input will always be a single number.
- You MUST use iteration on this problem:
  You have learned how to code something just like this using logical masking; however, this should be a simple function to learn the process of iteration.

Hints:

- Using a counter variable may be useful.

Drill Problem #2

Function Name: blackJackJr

Inputs:

1. (*double*) A vector of card values (digits 0-9)

Outputs:

1. (*double*) Total number of cards played *before* reaching a sum greater than 21

Function Description:

You are the oldest sibling of three on a family road-trip vacation. You are not ten minutes out of the city when your two younger siblings start fighting over which card game to play. Your brother wants to play a "grown-up" card game, but your sister is too young to understand what face cards and aces represent. So you, being the awesome older sibling that you are, keep the peace by creating a card game called Black Jack Junior.

Black Jack Junior is essentially the same game as regular Black Jack (keep playing cards to get the highest sum of equal to 21 or less) but played with your sister's Uno cards (with values 0 to 9). As the dealer, you know the order of the cards (the input to your function) and need to determine how many cards it takes to surpass 21.

Write a function in MATLAB called `blackJackJr()` that takes the input of card values (in order dealt), iterates through those values until the sum is greater than 21, and outputs the total number of cards played.

Notes:

- An additional condition of the game is to use as many cards as possible getting to 21 (and not over), so if your last card(s) is a 0, that is ok and it should be included in your total number of cards.
- You are guaranteed to surpass the total 21 within the given input vector of card values.
- You MUST use iteration in this problem.

Hints:

- Using a counter variable may be useful.

Drill Problem #3

Function Name: brickLayer

Inputs:

1. (*double*) The number of bricks to place vertically along the brick wall
2. (*double*) The number of bricks to place horizontally along brick wall
3. (*char*) Brick color

Outputs:

1. (*char*) An array of bricks

Function Description:

Georgia Tech's motto is "Progress and Service;" as part of progress, its plans to renovate the College of Computing has begun. As both a MATLAB guru and an aspiring architect, you are tasked with designing the brick walls of the building. Tired of the College of Computing's bland exterior, you decide to intersperse colored bricks in the new wall.

Given a string indicating a brick color, create an array of bricks comprised of alternating colored and non-colored bricks. To ensure structural stability of the brick wall, the corners of the brick wall must be colored bricks. The format of the colored brick is given as the following:

`'<space>[<input_color>]<space>'`

For example, if given the brick color blue, your brick to then iteratively lay would be represented by a string such as: `' [blue] '`. Non-colored bricks are the same size as colored bricks but contain dashes (Ex: `' [----] '`).

```
array = brickLayer(5, 9, 'blue')
```

```
array =>
 [blue]  [----]  [blue]  [----]  [blue]  [----]  [blue]  [----]  [blue]
 [----]  [blue]  [----]  [blue]  [----]  [blue]  [----]  [blue]  [----]
 [blue]  [----]  [blue]  [----]  [blue]  [----]  [blue]  [----]  [blue]
 [----]  [blue]  [----]  [blue]  [----]  [blue]  [----]  [blue]  [----]
 [blue]  [----]  [blue]  [----]  [blue]  [----]  [blue]  [----]  [blue]
```

Imagine how nice the College of Computing will look now! ☺

Notes:

- All bricks must be the same size, and each brick must be padded by a space on either of its sides. Therefore, two juxtaposed bricks should have *two* spaces between them.
- We will only provide an odd number for the number of bricks to place vertically or horizontally along the brick wall.

Hints:

- You may find a nested for loop useful for this problem.
- You may find it useful to nest conditional statements for this problem.

Drill Problem #4

Function Name: scottSterling

Inputs:

1. (*double*) A 2xN array containing kicked ball positions

Outputs:

1. (*double*) Scott Sterling's level of exhaustion after blocking so many shots

Function Description:

Step 1: Watch this week's Click of the Week (located at the bottom of the hw007.m file).
Step 2: Read below for instructions to write the MATLAB function `scottSterling()`.

Scott Sterling. The man. The myth. The absolute LEGEND. Scott Sterling, never one to back down from a challenge, has offered to help you with your MATLAB homework by providing his face and unflinching resolve to a simulated soccer shootout. In this shootout, balls will be kicked toward the goal and will pass the plane of the goal at x- and y-coordinates given in the input array. Your job is to iteratively calculate just how tired Scott Sterling will be after blocking every single shot that falls within the boundaries of the goal (including the goal border).

Assuming the goal to be 24 feet wide in the x direction and 8 feet tall in the y direction, and with the origin at the bottom left of the entrance to the goal, each shot must fall within these boundaries for Scott Sterling to even attempt to stop the shot. For the shots he does stop, Scott Sterling's exhaustion level increases for every 8 feet his face must travel to stop each ball (so exhaustion is equal to distance / 8). Scott Sterling's face will begin at the position (12, 6) and after each shot he blocks, his face will then be located at the position of the shot he just stopped. Ha ha. Good Show! Indeed.

Example: Corners of defined goal and Scott Sterling's face starting position:

[0, 8]		[24, 8]
	[12, 6] *Scott Sterling's face starts here*	
[0, 0]		[24, 0]

Notes:

- The input vector will contain x-values for each kick in the top row, and y-values in the bottom row.
- If the ball is kicked within the 24x8 (x-by-y) bounds of the goal (where shots on the boundary are included), Scott Sterling WILL block it. If it is not within these bounds, he will not move for that kick.
- The distance Scott Sterling has to move his face between each shot can be found by the same formula from the HW02 function `distCalc()`, you might recall.

Drill Problem #5

Function Name: passwordProtector

Inputs:

1. (*char*) A string representing your password

Outputs:

1. (*char*) An obfuscated version of your password

Function Description:

The alarms are blaring, there's gunfire from the hallway. Your plans for global domination seep away with each beep of the self-destruct sequence timer you've just initiated. You could split now—get in your enormous rocket and fly away to a different island lair—but you're no fool: you know your adversary, and if you don't make sure all your data is encrypted before you leave, your passwords will certainly fall into the hands of MI6. To do the encryption, you opt to write a quick MATLAB function.

In the function called `passwordProtector()` you will run the input string through a set of edits to be made. If the final string meets a certain condition you will output that final string as the protected password. If it doesn't meet that condition, you will run it through those same edits again (…think iteration!).

The editing algorithm to transform your password will run until the **average ASCII value of the string is equal to or greater than 160**.

The edits to be made each iteration are:

- Delete all spaces
- Replace all instances of capital `'A'` with the ASCII value 192
- Replace all instances of capital `'O'` with the ASCII value 212
- Replace all instances of lowercase `'e'` with ASCII the value 235
- Replace all instances of lowercase `'z'` with ASCII the value 191
- After all instances of lowercase `'b'`, insert the ASCII value 223
- Before all instances of capital `'F'`, insert the ASCII value 176
- Then, finally, shift all letters by 1 using `caesarCipher()`

As the last line of code is written but before you hit enter, the timer suddenly cuts off. You freeze and hear a click. Hard metal is pressed against the back of your head as some narcissistic spy tells you his name, repeating part of it for emphasis: 'Bond…James Bond.'

Notes:

- The savvy coder will notice that it is possible to have a password in which the terminating condition is never met. We will NOT test you against such cases. You are guaranteed to reach the terminating condition for all input passwords.

Drill Problem #5

- The `caesarCipher()` function ought to be used as a helper function (or you can rewrite your own `caesarCipher()` function). Note that `caesarCipher()` itself used `caesarSalad()` as its own helper function.

Hints:

- The built-in function `strfind()` will prove incredibly useful.

Drill Problem #6

Function Name: templeOfTime

Inputs:
1. (*double*) A 1x2 vector representing a row and column
2. (*char*) An MxN string array representing cardinal directions
3. (*double*) An MxN array representing distance to travel
4. (*char*) An MxN string array of jumbled letters
5. (*double*) An MxN array of rupee values

Outputs:
1. (*char*) A string found after following the directions through the array of letters
2. (*double*) The total number of rupees collected after travelling through the array

Function Description:

In the Legend of Zelda games, it is usual to be challenged with a series of puzzles (or evil creatures) in order to advance between rooms in dungeons, find keys, and unveil treasure chests with the next dungeon weapon (or that oh so useful compass…). While in the most recent temple, Link (the main character) has found a scroll infinitely more useful than the mere map other dungeons provide.

Examining the scroll, he sees a vector of two numbers; an array of the letters `'N'`, `'S'`, `'E'`, and `'W'`; an array of seemingly random integers; another array of jumbled letters, this time spanning the entire alphabet; and an array of rupee values (rupees being the form of currency in the game). Navi, Link's fairy, gives an essential hint ("this scroll is probably useful for this dungeon"), and Link realizes that the scroll is a puzzle! It is a set of instructions to find what the dungeon's weapon is and where it is located.

The vector of two numbers are Link's starting coordinates in the arrays. The first array of the letters `'N'`, `'S'`, `'E'`, and `'W'` are cardinal directions through which to traverse the arrays. The first set of numbers is the distance to travel in any given turn. The second char array is a set of jumbled letters that—when formed in the correct combination—spells the weapon that may be found in Link's current dungeon. Finally, the last array contains values for rupees that Link picks up along the way, since there is money laying around everywhere in this dungeon.

Write a function called `templeOfTime()` that follows the pattern given by the clues in the input cardinal direction array and travel distance array to output a string indicating the dungeon's weapon and a double of the total number of rupees found in the dungeon. The first input is a 1x2 vector representing the starting index. This is the location of the first letter in the string array (fourth input) and where to begin in all four input arrays.

Next, using the second input (the direction array) and the third input (the distance travelled array), grab the values in each array corresponding to the current index location. The direction will be designated in the second input by `'N'`, `'S'`, `'E'`, or `'W'` for north (up), south (down), east (right), or west (left). Find the number of steps to be taken in the third input, and travel that many spaces in the direction given. The number of steps to be taken will always be positive.

Drill Problem #6

Trace through all the arrays simultaneously, using the directions and distances as a guide. Save the letter found in each step (from the array in the fourth input) into a string for the final dungeon weapon answer. Further, keep track of the total number of rupees collected along the trip—the number of rupees being an array given in the fifth input. Include the letter and rupee value found in the starting location. Stop when the direction is `'D'` (for destination) and the number of spaces is 0.

For example:

If the inputs were the following:

```
[1,3] , ['NSEW;    ,  [4 1 1  2;  , ['ZiSh;  , [ 1 5 50 1;
          DEWW']       0 1 2 18]       del ']    10 5  5 1]
```

The starting index would be the first row and third column (1,3) in the fourth input (`'S'`); the fifth input would give 50 rupees to start with. Then look at index (1,3) in the second input (`'E'`) and in the third input (1). This gives directions to go to the index 1 to the right of the current position; this makes the updated position now (1,4), which holds `'h'` in the fourth input and 1 rupee. Now the stored string should read `'Sh'` and the total rupees be 51. To find the next index, look at the position (1,4) in the second and third inputs. Follow the pattern until the letter `'D'` is found in the direction array and 0 is in the travel distance array. The output string of this example is `'Shield'` and the total rupee amount is 76.

Notes:

- The string array holding the password may include non-letter characters (i.e., `' '`, `'.'`, `'?'`, etc.), but an apostrophe will not appear.
- The correct path will not take you out of bounds of the array, but a wrong one might.
- Once the direction is `'D'` and the number of spaces to travel is `0`, you should stop traversing the dungeon.