Congratulations on finishing a semester of CS 1371 Homework! You have made it to the final extra credit assignment. These problems are meant to be somewhat challenging and also to help you review different concepts from throughout the semester. There are six problems, each worth around 20 points. There are also two ABCs, which contain some conceptual practice problems about sorting, big O, and graph theory to help you prepare for the final. So doing all six problems could get you 120 points of extra credit towards your homework average! This homework is due on the very last day of class, **Tuesday December 4th** at 8:00 pm, with the usual grace period until 11:59 pm. Since it is due on the last day of class, there will not be a resubmission. Have fun with these problems, good luck with finals, and thanks for a great semester in CS 1371!

Happy Coding!
~Homework Team

Function Name: `collatz`

Inputs:

1. (*double*) Any positive integer

Outputs:

1. (*double*) The number resulting from the algorithm
2. *(double)* The number of recursive steps required by the algorithm

Function Description:

The Collatz conjecture (also known as the 3n+1 conjecture) was proposed by Lothar Collatz in 1937. The conjecture says that any positive integer n can be recursively manipulated **to be a number less than 2** by the following algorithm:

1) if the number is even, divide it by 2
2) if the number is odd, multiply it by 3 and add 1
3) repeat (i.e. recursively call the function)

Your job is to write a recursive MATLAB function that implements the Collatz conjecture. Your function should output the final number that the algorithm reaches as well as the number of recursive calls it took to get there.

Notes:

- If you try running this function with very large numbers (as in 40-digit numbers), MATLAB will crash as it will reach its maximum recursion limit.

Hints:

- You may find it useful to make a helper function to keep track of the number of recursive calls that have been made.

Function Name: `maskingTape`

Inputs:

1. *(char)* Filename of the image of your unpainted wall
2. *(char)* Filename of the image showing what you want your wall to look like
3. *(double)* 1x3 vector representing the RGB values of the masking tape color

Outputs:

(none)

File Outputs:

1. Image of masking tape applied to original image

Function Description:

You are thinking of sprucing up your dorm room by painting your walls, and decide to use MATLAB to help you out! You are given an image of an unpainted wall, and an image of what you want the painted wall to look like. Based on these two images, determine where masking tape must be applied to the unpainted wall so that the appropriate areas remain unpainted.

Create a new image showing what the unpainted wall looks like with masking tape applied. The color of the masking tape is given by the third input, with the three values in the vector representing its red, green, and blue values, respectively. The new file name should be the same as the first input, with `'_tape'` appended before the file extension.

Notes:

- The painted wall may be painted with any assortment of colors.
- Any part of the painted wall that is the same color as the unpainted wall will be covered in masking tape.

Function Name: `warriors`

Inputs:

1. (*struct*) An 1xN structure array
2. (*char*) A field name

Outputs:

1. *(struct*) An 1xM updated structure array
2. (*char*) A descriptive statement about the clan battle

Background:

While reading everyone's favorite childhood book series about clans of feral cats, *Warriors*, you realize that the plot is quite difficult to keep up with at times. Rather than simply following the exciting adventures of the protagonist, Firepaw of the ThunderClan, like a casual reader, you decide to go the extra step. Since you are such a dedicated fan of this incredible series, you will write a MATLAB function to track the battles of the clans throughout the saga.

Function Description:

Write a function that takes in an 1xN structure array and a field name of a statistic that is the most important in a particular battle. Each structure in the array represents a different clan of cats. Based on the values of the field from the given field name (second input) which are guaranteed to be of type double, you will determine which clans battle and who will win. The clan with the highest value in the given field will win in a warrior battle against the clan with the lowest value in the given field. Any intermediary clans will not be affected by the battle. Make the following updates to the structure array based on this battle:

1. The winning clan's structure's value in the field given by the second input should be doubled.
2. The winning clan's structure's value of the `Territories` field, represented by a cell array of strings, should be updated to include the values of the `Territories` field in the losing clan's structure.
3. The losing clan's value in the field given by the second input should be set to zero.
4. The losing clan's value for the `Territories` field should become an **empty cell array**, `{}`.
5. Sort the entire array in descending order based on the values in the field given by the second input. Do this using the `'descend'` input to sort.

In addition to updating the structure array, the function should also output a descriptive statement in the following format:

```
'Following the warrior code, fearless leader <winning leader> led
    <winning clan> to victory against <losing clan>.'
```

Continued...

Example:

```
>> clans =
```

Name : 'ShadowClan' Leader : 'Brokenstar' Territories : {'northeast marshes'} Strength : 75	Name : 'ThunderClan' Leader : 'Firestar' Territories : {'southeast woodlands'} Strength : 80

```
>> [updatedClans, result] = warriors(clans, 'Strength')

updatedClans =
```

Name : 'ThunderClan' Leader : 'Firestar' Territories : {'southeast woodlands', northeast marshes} Strength : 160	Name : 'ShadowClan' Leader : 'Brokenstar' Territories : {} Strength : 0

```
result = 'Following the warrior code, fearless leader Firestar led Thunderclan to victory against Shadowclan.'
```

Notes:

- The input structure array is guaranteed to have the following fields: `'Name'`, `'Leader'`, and `'Territories'`.
- It is guaranteed that the values in the field given by the second input will be a 1x1 double.
- It is guaranteed that the value in the `Territories` field will be a cell array.
- There will not be any ties.

Function Name: `recipe`

Inputs:

1. (*char*) The filename of an excel sheet containing grocery store inventory information
2. (*char*) The filename of a text file containing a recipe

File Outputs:

1. A text file containing directions for the shopping trip

Background:

You are preparing to host a huge dinner party for all of your friends and family when you realize you have absolutely none of the ingredients you need! The party is in a few hours, and you don't have time to manually go through all the recipes and figure out what to buy for the dinner party. Luckily you have MATLAB to help!

Function Description:

Write a function that takes in a text file of a recipe and extracts the needed ingredients. You should look for these ingredients in the excel spreadsheet of the grocery store inventory to figure out how much to buy, and print that out to a new text file. You also need to add up the total cost of the trip, and write that in a statement to the last line of the text file.

All of the ingredients in the recipe will be given as either 'cup', 'pound', 'teaspoon', 'tablespoon', 'package', or just a single number. All of the items in the grocery store are sold in terms of 'oz' or 'count' (i.e. the price of 1 count tomato, the price of 12 count eggs). You must make the necessary conversions as follows:

- If the item on the ingredient list is in terms of 'cup', you need to get 8 oz of the item at the store
- If the item is in terms of 'pound', you need at least 16 oz from the store.
- If the item is in 'teaspoon' or 'tablespoon', it is safe to assume that one package will be enough, and you only need to buy one unit sold at the grocery store
- If the item lists 'package' or it doesn't specify (no units), buy the appropriate number of units

Use this conversion to figure out how many packages you would need to buy from the store. Since it's a store, you can't split up the packages. So if you needed 14 eggs, and each package only contains 12 eggs, you would have to buy 2 whole packages.

Once you know how much to buy, print the directions to a new text file. The name of the text file should be the name of the recipe text file with `'_list.txt'` appended to the end.

...Continued

For each line of the new text file, there are 2 possibilities. If the recipe does not specify units, the line should read `'Get <number to buy> <item> at $<price> each.'`. If units are specified, the line should read `'Get <number to buy> package(s) of <item> at $<price> each.'`

After all the ingredients needed are written to the new text file, add a line at the end that reads `'Total cost of the trip: $<amount spent>'`

Example:

```
      From Recipe:                  From Grocery Store inventory
      (3 cup) Sour Cream            {Sour cream} {1.98} {16 oz}

→ Written to new text file:
      'Get 2 package(s) of Sour Cream at $1.98 each.'
```

Notes:

- Each recipe is formatted with Ingredients listed first, followed by directions, indicated by a line `‘Directions:’` after the ingredients.
- On the ingredient list, the amount of the ingredient will be listed in between two parenthesis, followed by the item name.
- The grocery store has the name of the item in the second column, the price in the third column, and the amount per package in the 4th column
- Matching case is not guaranteed, however you don’t need to account for different spellings or forms of the word (i.e. cup vs. cups, potatoes vs. potato)
- The units of the grocery store will correspond to the desired units from the recipe. For example, if the recipe calls for 2 cups of an item, the grocery store will sell the item in terms of oz.

Function Name: `integrals`

Inputs:

1. (*double*) A 2xN double array representing (x,y) coordinates, one point per column
2. (*double*) A 2xN double array representing (x,y) coordinates, one point per column

Outputs:

1. *(double*) The area between the input functions

Plot Outputs:

1. A plot of the two functions with filled in area

Background:

You have been collecting data on two runners, keeping track of their speed by taking measurements at discrete points in time. In order to confirm the speed readings, you need to integrate and compare the final difference between the two to the distance between them when they stopped.

Function Description:

Given two sets of (x,y) values ordered by their x values, compute the positive area between the two curves. One set of points is guaranteed to lie entirely above the other, but it may be either the first or second input. In other words, the y-values of one input will be greater than the corresponding y-values in the other input. Use the trapezoidal approximation of the integral to find the area under each, and take the positive difference.

We need a visualization for the math we just did! First, plot the greater of the two inputs with a green line, and plot the lesser of the two with a red line. Next, 'fill in' the area between the two curves you just plotted with a vertical line of magenta stars at every integer along the x-axis (`'m*'`) using the following steps:

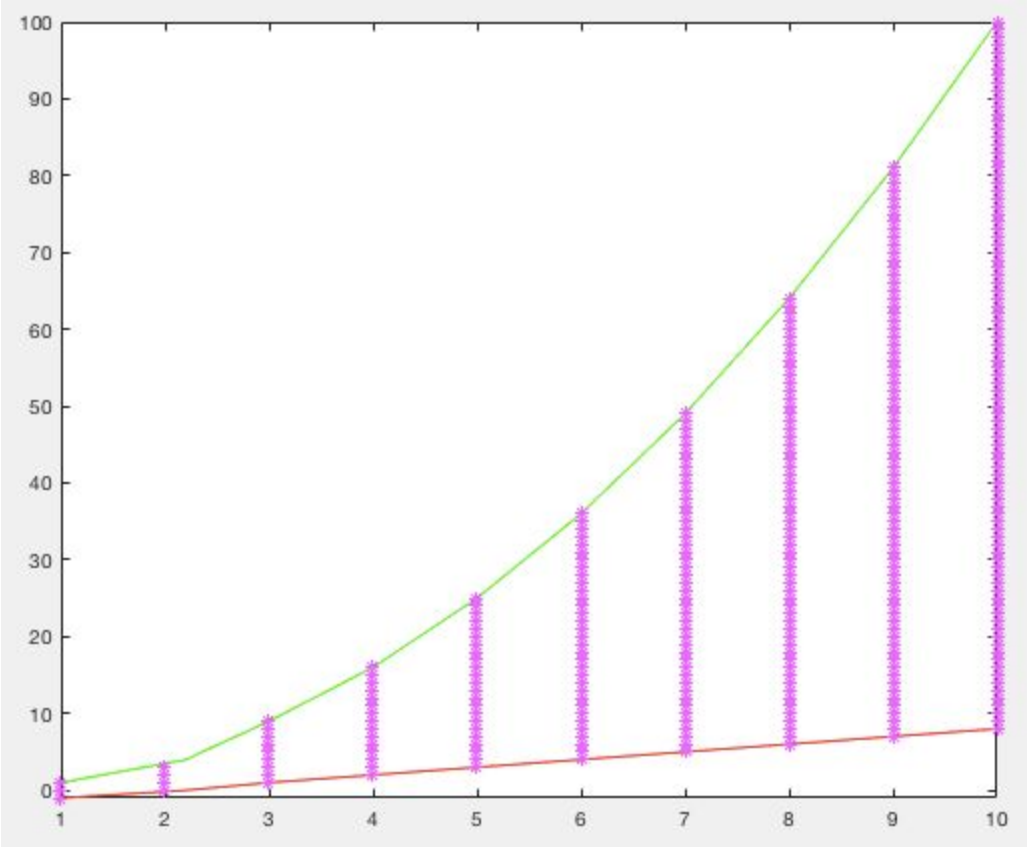
1. For every integer in between the minimum x value to the maximum x value, linearly interpolate the corresponding y values on the lower and upper curves.
2. At each x value, plot the stars from the interpolated y value on the lower line to the interpolated y value on the upper line so that each star will lie at an integer point.

Example:

```
>> curve1 = [1   2.2   3   4   5   6   7   8   9   10;
             1    4    9   16  25  36  49  64  81  100]

>> curve2 = [1  2.2  3  4  5  6  7  8  9  10;
            -1   0   1  2  3  4  5  6  7   8]

>> area = integrals(curve1, curve2)
area = 302.4
```



Notes:

- *After* you finish plotting, adjust the axis limits to be as small as possible to contain all data points.
- Round the **final** area output to two decimal places.
- Use `ceil()` and `floor()` when determining where to plot the lowest and highest stars in each vertical line, respectively.
- You are guaranteed that the two inputs will have the same domain; one set of points will not extend past the other.
- The smallest and largest x-values will be integers, so you do not need to worry about rounding these values when plotting the magenta stars.

Hints:

- When finding which y values to plot at each x value, think about how can you use the colon operator.

Function Name: `imitationGame`

Inputs:

1. *(char)* Input message
2. *(char)* 26x(2N) array of rotor settings
3. *(char)* Mx2 array of plugboard wirings

Outputs:

1. *(char)* Output message

Background:

You are a British codebreaker working at Bletchley Park during World War II. Your task is to help the British government crack the Enigma machine, which Nazi Germany used to encode messages. While your co-worker Alan Turing works on his Bombe machine, you take a different approach. You've developed a handy MATrix LABoratory (MATLAB) that you believe can help turn the tide of the war. Thanks to some Polish cryptographers, you have a pretty decent understanding of how the German Enigma works, so you decide to create a simulation of it in MATLAB!

Function Description:

The Enigma machine was designed to encrypt a message one letter at a time. First, the letter was typed into a machine. The letter went to the plugboard, where it could be switched with another letter or remain the same. It then went through a series of rotors, each of which swapped letters in a fixed pattern. It then went through the rotors in reverse, then back through the plugboard, to get the encrypted letter. After each letter, the first rotor would shift by one, meaning that typing the same letter twice in a row resulted in two different letters output. After 26 rotations of the first rotor (i.e all the way around), the second rotor would shift by one, and after the second rotor rotated 26 times (and the first rotated 26 * 26 = 676 times), the third rotor would shift. The last rotor, known as a reflector, did not rotate, but simply linked two letters together. This made Enigma very difficult to crack.

Write a function that takes in an input message, a character array describing the rotors, and a character array describing the plugboard wirings. If there are N rotors, including the reflector, the rotor array will be 26 x (2N). The first two columns of the array describe the first rotor. Each row represents a letter and what it is mapped to. For example, if a row is `'AD'`, then when a letter A is input into the rotor, it comes out as D. If going through the rotors in reverse, then an input of the letter D is mapped to A. The next rotor is represented by the next two columns. The last two columns represent the reflector. A rotation of the rotor is modeled by shifting the second column of each rotor down by one, so that the element in the first row becomes the second and the last element becomes the first.

The M plugboard wirings are represented by a Mx2 character array. Each row represents two letters that are connected to each other. For example, if a row is `'DZ'`, then D becomes Z and Z becomes D when going through the plugboard.

Example:

`imitationGame('aa',rotor1, ['ad';'cn'])` (`rotor1` is the one used in the test cases) =>

The first letter we wish to encode is `'a'`. We set the encrypted letter to be `'a'`. One of the rows in the wiring is `'ad'`, so we set the encrypted letter to be `'d'`. `rotor1` is 26x6, so we know there will be two rotors and a reflector. In the first two columns of `rotor1`, we see that one row is `'dp'`. Thus, the encrypted letter becomes `'p'`. In the third and fourth columns, we see that one row is `'pc'`, Thus, the encrypted letter becomes `'c'`. In the last two columns (the reflector), we see that one row is `'ca'`, so the encrypted letter becomes `'a'`. In the third and fourth column, we see that one row is `'ua'`, so the encrypted letter becomes `'u'` (note that this is in reverse!). Finally, in the first two columns, we see the row `'ku'`, so the encrypted letter becomes `'k'`. None of the plugboard wirings have the letter `'k'`, so the first letter in the output is `'k'`.

We rotate the second column by one unit downward (the first row is now the second, the second the third, etc.) The letter we once again wish to encode is `'a'`, so the encrypted letter is `'a'`, then `'d'`, as in the first letter. Now however, the relevant row in the first two columns is `'dc'`, so the letter becomes `'c'`. Then, in the third and fourth columns (which are unchanged!), we see the row `'ch'`, so the letter becomes `'h'`. In the reflector, we see the row `'hm'`, so the letter becomes 'm'. In the third and fourth columns, we see the row `'mz'`, so the letter becomes `'x'`. In the first and second column (second is shifted), there is the row `'zn'`, so the letter becomes `'n'`. One of the wirings in the plugboard is `'cn'`, so the letter becomes `'c'`. Thus, the output is `'kc'`.

Notes:

- Do not hardcode for the number of rotors or the number of wirings!
- Each letter only passes through the reflector once.
- The reflector never rotates.
- Note that the example is not the same as the first test case. You can still use the solution to verify the example provided.

Hints:

- You will need to think of a way to remember when to rotate the rotors.
- One flaw of the Enigma, which was used to help crack the machine, is that no letter can be encoded to itself. If your machine encodes a letter to itself, something has gone wrong!
- The `circshift()` function will help you shift the rotors.