Notice

This homework is designed to cover various topics from throughout the semester. It is for extra credit as well as to assist you in preparing for the final exam. However, it does not exhaustively cover all topics that you need to know for the final. You have until Tuesday, December 5th to submit it. To confirm, this homework is not mandatory and is only for extra credit! There are a total of 160 extra credit points possible (20 points per problem).

Thanks for a great semester!
Good luck on the Final,
and as always,

Happy coding,
~Homework Team

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.

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: `movieStar`

Inputs:

1. *(struct)* A 1xN structure array of movie data

Outputs:

1. *(char)* The highest paid actor/actress
2. *(double)* The total amount of money he/she made
3. *(double)* The average movie rating of movies he/she appeared in

Function Description:

As a self-described movie buff, you enjoy spending your time doing all kinds of research about the film industry. This week, you've decided to do some research about how much actors and actresses are paid, using data from a structure array that you found on the internet.

The structure array will always have the following fields, and may have other fields.

Field name	Data within field
`Bad_Vegetable_Score`	A decimal percent between 0 and 1
`Budget`	A double
`Revenue`	A double
`Cast`	An Mx1 cell array of cast member names

An example structure can be seen below:

```
sa(1) =>
      Movie_Name: 'Puzzles'
      Year: 1984
      Genre: 'Mystery'
      Run_Time: '1h23m'
      Director: 'Kim Coppola'
      Budget: 5000000
      Revenue: 5500000
      Profit: 500000
      Cast: {'John Cena';
             'Stew Leonard';
             'Sam Walton';
             'Aunt Jemima';
             'Betty Crocker'}
      MPAA_Rating: 'PG-13'
      Bad_Vegetable_Score: .32
```

To calculate the highest paid actor, you should assume that the profit from the movie is split evenly between all of the actors/actresses in the movie. The profit of the movie is the difference between the budget and the revenue. If the movie lost money (revenue less than budget), then assume none of the cast members made any money. Once you have determined the highest paid actor/actress, you need to determine the average rating of the movies he/she was in. However, because you are assuming the highest paid actor/actress is a self-respecting individual, you should remove all movies in which Nicolas Cage is a co-actor before calculating the average movie rating. Additionally, if Nicolas Cage is the highest paid actor, you should assume there is a mistake in the data and output values for the second highest paid actor/actress.

Notes:

- The actor/actress money should be calculated before removing Nicholas Cage.
- An actor or actress' name will always appear exactly the same way in any movie in which they acted. Nicolas Cage will always appear as **`'Nicolas Cage'.`**
- An actor or actress will appear in the cast section only once per movie.
- In the case of a tie for highest paid actor/actress, select the actor/actress whose name appears first alphabetically.
- You should round every money calculation to two decimal places.
- Round the average rating output to two decimal places.

Function Name: `dragRace`

Inputs:

1. *(double)* An NxM array of sampled points in time
2. *(double)* An NxM array of sampled velocities
3. *(cell)* A 1xN array of car names
4. *(double)* The specified race distance

Outputs:

1. *(char)* A string describing the race results

Function Description:

You always dreamed of being a racecar driver, but with your numerous other MATLAB projects, you haven't had the time to practice. Fortunately, the most prestigious auto race in the world is taking place this week: The MATLAB Grand Prix!

Using the given set of times and velocities, determine which car wins the race of given length and which car has the fastest acceleration. Each row corresponds to the data of one racer whose name is in the corresponding index in the cell array input.

Numerically integrate the given values to find the distance each car covers. Since each racecar will likely cross the finish line between two sampled points, linearly interpolate to find the times each car crosses the finish line. Then, numerically differentiate the velocities and determine which car has the fastest acceleration between two adjacent data points.

Output a string with the format:

```
'The <race winner> won the <race distance> meter race in <time> seconds! The <fastest acceleration car> had the fastest acceleration at <acceleration> m/s^2!'
```

Numbers in the output should be rounded to the nearest tenth. Be sure to check your answers against the solution file.

Notes:

- The times are given in seconds and the velocities are given in meters per second.
- The times and velocities array will have the same number of columns.
- The number of cars and the number of rows in the times and velocities arrays are the same.
- Remember to use linear interpolation between data points.
- You can use `'%0.1f'` as a format specifier inside of `sprintf()` to display a number to one decimal place.

Hints:

- `cumtrapz()` will be useful.

Function Name: **`microscope`**

Inputs:
1. *(char)* The name of a microscope image
2. *(double)* The 'scale' represented by the scale bar

Outputs:
1. *(double)* A 1x2 vector for the dimensions of the microscope image

Banned Functions:

`rgb2gray`, **`im2bw`**

Function Description:

Published microscope images often have a black scale bar to indicate the scale of the image. Write a function called **`microscope`** which will determine the actual dimensions of the image based on the the relative size of the scale bar to the entire image. The second input to the function is the actual unitless distance represented by the scale bar.

For example, given the 400x500 image example.png with a scale bar length of 100 pixels,

example.png (400x500)

`[dim] = microscope('example.png', 50)` should return **`[200 250]`**, since the height of the image can fit 4 scale bars (200 units of length in total), and the width of the image can fit 5 scale bars (250 units of length in total). You should round your final output to the nearest integer.

Notes:
- The scale bar may be oriented horizontally or vertically; the longer side corresponds to the length input.
- The scale bar will be the only pure black pixels on the image.
- The output dimensions should be [height width].

Function Name: `snowflake`

Inputs:

1. *(double)* The recursive depth

Outputs:

(none)

Plot Outputs:

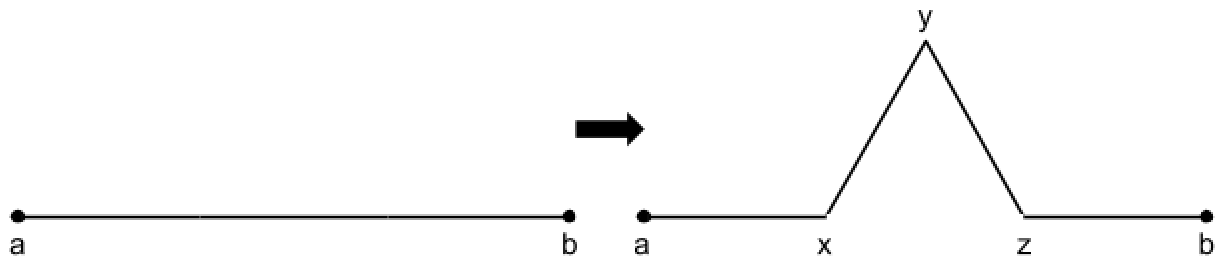
1. A plot of the Koch Snowflake

Function Description:

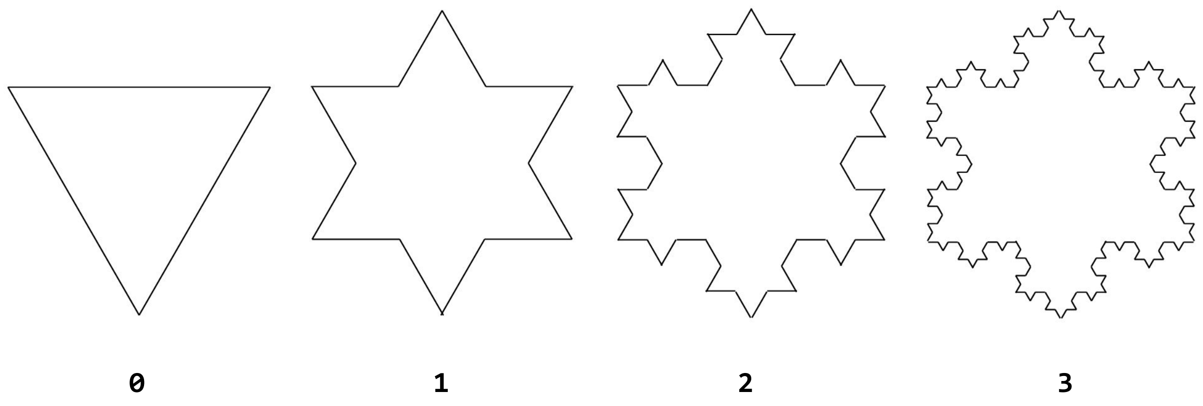
Fall's got you feelin some type of way, but what you're really looking forward to is snow! But since it never snows in Atlanta, you're going to have to settle for a MATLAB function that plots snowflakes instead.

You may or may not have heard of fractals before, but they are incredibly complex structures that occur everywhere in nature. From trees to computer networks to city planning, fractals have been used to model all sorts of things. You can read more about them here.

For this problem you will be creating a plot of a well-known fractal, namely, the Koch Snowflake. The Koch Snowflake is created by starting with an equilateral triangle and recursively replacing every line segment with 4 line segments as follows:



Line segments $\overline{ax}$, $\overline{xy}$, $\overline{yz}$, and $\overline{zb}$ are all the same length. Additionally $\triangle xyz$ is an equilateral triangle. Based on this rule, the first few recursions of the Koch Snowflake look like this:



The first input to the function determines how many times you should recurse. So **`snowflake(0)`** will be the first triangle pictured above, **`snowflake(3)`** will be the snowflake on the right and **`snowflake(6)`** will have even more detail. The top left, top right and bottom point of the starting triangle should be **`(-1, 0)`**, **`(1, 0)`** and **`(0,`** $-\sqrt{3}$ **`)`**, respectively.

In case your geometry is a little rusty, you have been provided with a helper function, **`triPoints()`**, that will produce the coordinates for points x, y, and z given points a and b. You can read more on how to use this function by typing help triPoints in the Command Window.

Notes:

- Use **`axis equal`** and **`axis off`**.
- Plot in a black line.
- Because of the computational complexity of plotting the Koch Snowflake, you will probably not want to test your code on an input larger than 12 or 13. The Koch Snowflake with a recursive depth of 13 has over 200 million line segments. With a recursive depth of 19 there will be more line segments than there are stars in the Milky Way!

Hints:

- Focus on creating one side of the triangle first, then extend your idea to the remaining sides.

Function Name: `snekingZoo`

Inputs:

1. (*struct*) an MxN structure array representing a zoo

Outputs:

1. *(struct*) MxN structure array representing the sorted zoo

Background:

Remember **`remodelZoo`** from the structures homework? Well, It's back and better than ever! The very grateful Atlanta Zoo implemented your **`remodelZoo`** function which boosted revenue! However, they still are not maximizing their profits and need your help to remodel the zoo again!

Function Description:

Much like **`remodelZoo`**, you are given MxN structure array representing a zoo. Each row represents a section of the zoo (i.e. reptile house, primate pen, etc.), and each column represents a specific exhibit (i.e. salamander, komodo dragon, iguana, rattlesnake, etc.). Each exhibit has the following fields that contain the subsequent information: **`'exhibit'`**: *(char)* the name of the animal in the exhibit, **`'people'`**: *(double)* the average number of people who visit the exhibit per hour*,* **`'time'`**: *(double)* the average time, in hours, each person spends in the exhibit, and **`'rating'`**: *(double)* the average rating (out of 5 stars) visitors gave the exhibit.

Your first job is to update each exhibit with a popularity score under a new field called **`'popularity'`.** You are given the helper function **`calcPopularity`**, which will **only return the popularity value.** You will have to add the **`'popularity'`** field to each index in the zoo yourself. To find out how this new version of the function works, type **`help calcPopularity`** into the command window. Your second job is to reorganize the zoo such that the **least** popular section is in the first row and the most popular is in the last. You should also reorder the exhibits in each row so that rows with odd indices are sorted from most popular to least popular and ones with even indices are sorted from least popular to most popular. The result, will be a *sneking* path that takes zoo goers from the most popular to least popular exhibits in each section (alternating left to right on first row, right to left on second row, etc.) and from the least popular section to the most popular section.

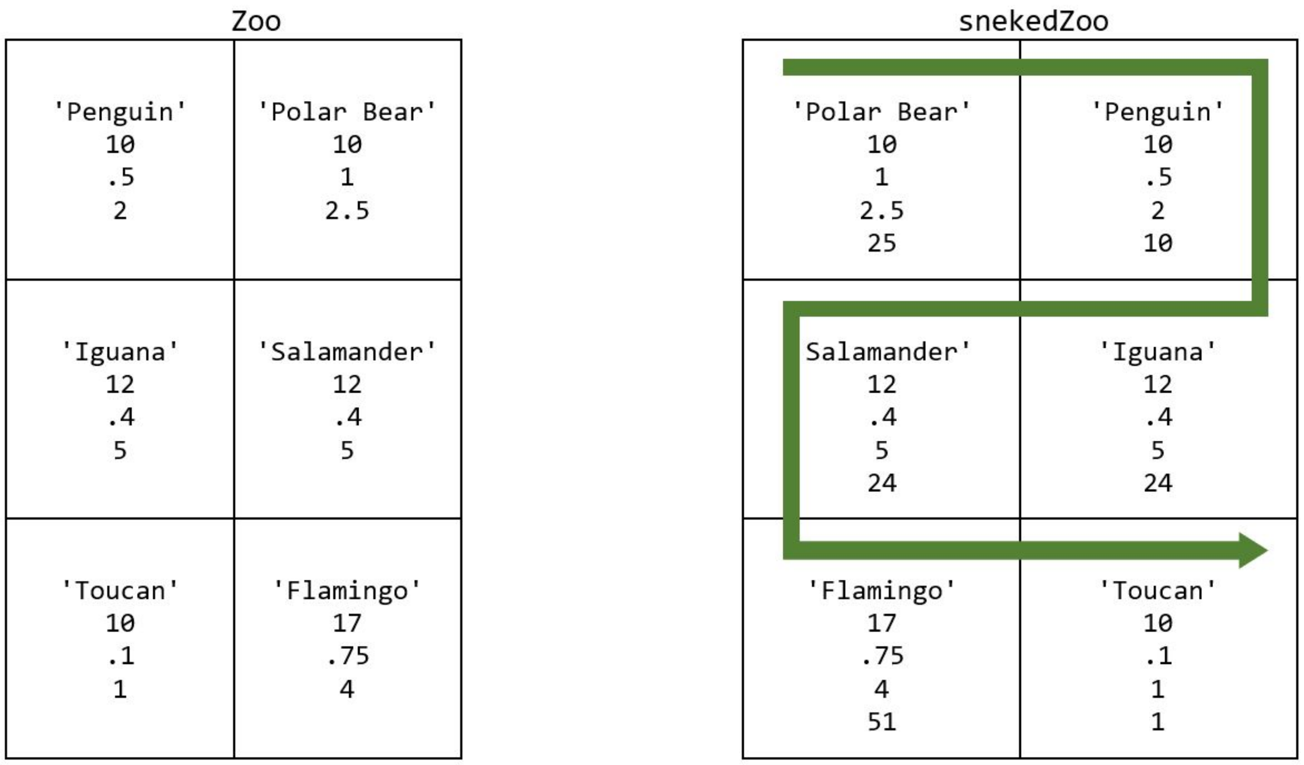
You are also given a helper function called **`displayZoo`** which will display a zoo in the command window. To see exactly how this function works and what it outputs, type **`help displayZoo`** in your command window. Please remember this is a tool to help you debug your code and figure out the values stored in different fields, and is not sufficient to test if your ouptut equals the solution file's output

Continued…

Example:

For the following example, each box represents an index in the structure array and each line represents a different field: exhibit, people, time, rating, and popularity respectively.

Given the input **`Zoo`** (where each section is read left-to-right), your function should output **`snekedZoo`** and follow the sneking path outlined in green.



Zoo

'Penguin' 10 .5 2	'Polar Bear' 10 1 2.5
'Iguana' 12 .4 5	'Salamander' 12 .4 5
'Toucan' 10 .1 1	'Flamingo' 17 .75 4

snekedZoo

'Polar Bear' 10 1 2.5 25	'Penguin' 10 .5 2 10
Salamander' 12 .4 5 24	'Iguana' 12 .4 5 24
'Flamingo' 17 .75 4 51	'Toucan' 10 .1 1 1

Notes:

- The total popularity of a section can be found by summing the popularities of each individual exhibit.
- If two sections have the same overall popularity, then they stay in the same order as the original zoo
- If two exhibits have the same overall popularity, then they stay in the same order with respect to the sneking path. (look at the second row of the example)
- Using sort with the **`'descend'`** input is different than reversing the second output to regular sort

Hints:

- Try and remember how to sort structure arrays based on a specific field!
- You already have code to sort the rows of a zoo, you just may have to modify it for this problem

Function Name: **`ultimateTicTacToe`**

Inputs:

1. (*char*) A MxN character array representing an ultimate tic-tac-toe board with the first move already made
2. (*double*) A 2xP array containing a sequence of moves to make

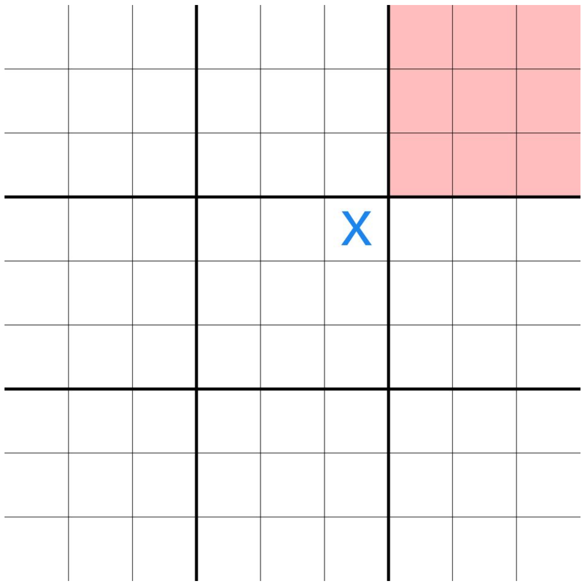
Outputs:

1. (*char*) The MxN board after the moves are made
2. (*char*) A string describing the result of the game

Background:

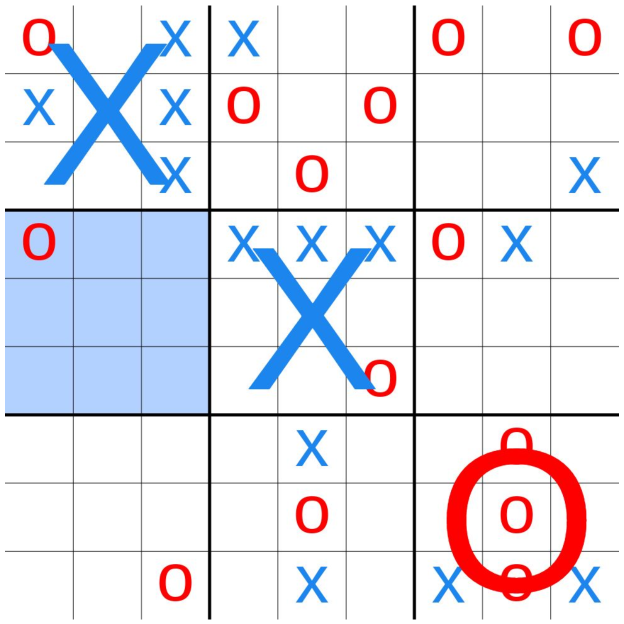
You've probably heard of regular tic-tac-toe, but playing the game perfectly is pretty easy; two experienced players will usually always tie because of the simplicity of the game. In fact, the game is strongly solved, which means that given any position, the outcome of a game can be exactly predicted if both players play perfectly. To enter the ranks of the MATLAB legends and truly become one of the greats, coding regular tic-tac-toe isn't going to cut it. It's time to get **ultimate**.

Ultimate tic-tac-toe is similar to regular tic-tac-toe, but instead of each square in a 3x3 grid having a single box, each square becomes a tic-tac-toe board, so an ultimate board looks like this:



After each move, the location of the previous move determines which board the next player must play on. In the picture above, player X played in the top-right corner of the center square, which means that player O must play in the top-right board (highlighted in red) of the overall board.

Just like in regular tic-tac-toe, if a player wins one of the small boards, that player will win that square in the overall board. If a small board results in a tie, then no player can claim that board. Once a board is completed, no more moves can happen there. If a player is sent to a board that has already been won, then that player can choose any board to play on. (For simplicity, this behavior is changed slightly for this problem - see the notes section.) The goal is to win three boards in a row diagonally or horizontally. A game in progress might look like this:



A more in-depth explanation can be found here, and you can try playing the game for yourself here.

Function Description:

Your job is to implement a function in MATLAB that takes in an ultimate tic-tac-toe board of any size and simulates the game, according to the rules for ultimate tic-tac-toe and the array of moves given.

The tic-tac-toe board given as the input to the function is a MxN character array representing the board. Empty boards may look like this:

```
=========================
|_|_|_|  |_|_|_|  |_|_|_|
|_|_|_|  |_|_|_|  |_|_|_|
|_|_|_|  |_|_|_|  |_|_|_|

|_|_|_|  |_|_|_|  |_|_|_|
|_|_|_|  |_|_|_|  |_|_|_|
|_|_|_|  |_|_|_|  |_|_|_|

|_|_|_|  |_|_|_|  |_|_|_|
|_|_|_|  |_|_|_|  |_|_|_|
|_|_|_|  |_|_|_|  |_|_|_|
=========================
```

```
==========================================
|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|
|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|
|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|
|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|

|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|
|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|
|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|
|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|

|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|
|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|
|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|
|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|

|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|
|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|
|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|
|_|_|_|_|  |_|_|_|_|  |_|_|_|_|  |_|_|_|_|
==========================================
```

Although tic-tac-toe is traditionally 3x3, your function should work for boards of any size. The rules for K by K (where K > 1) tic-tac-toe are the same as for traditional tic-tac-toe: K of 'X' or 'O' in any direction (vertical, horizontal, or diagonal) is a win. For example, the rightmost image above represents a 4 by 4 ultimate-tic-tac-toe board, which comprises a 4 by 4 grid of 4 by 4 tic-tac-toe boards.

The second input will be an array of moves, where each column represents a move for the current player. Since the board will start with one player having already gone, the first move in the array will be the move for the other player. The row index of the move is in the first row and the column index of the move is in the second row. These indices are relative to the <u>local board</u>, not the overall board, so it's up to you to figure out which square in the overall board each move should take place in.

When a board is won, fill it with an ASCII art version of that player's character, made to be the same size as the board (K by K). The following table lists examples for several board sizes, but since any board size may be given as an input, you should adapt the pattern to any board size.

Board Size (K by K)	Winning Boards

3x3	`|x| |x|`
	`| |x| |`
	`|x| |x|`
	
	`|o|o|o|`
	`|o| |o|`
	`|o|o|o|`

4x4	`|x| | |x|`
	`| |x|x| |`
	`| |x|x| |`
	`|x| | |x|`
	
	`|o|o|o|o|`
	`|o| | |o|`
	`|o| | |o|`
	`|o|o|o|o|`

5x5

```
|X| | | |X|
| |X| |X| |
| | |X| | |
| |X| |X| |
|X| | | |X|


|O|O|O|O|O|
|O| | | |O|
|O| | | |O|
|O| | | |O|
|O|O|O|O|O|
```

Output the board after all the moves have taken place, along with a string describing the output of the game. Return the string **`'<player> won.'`** if somebody won, **`'It's a tie.'`** if the game tied, and **`'The game's not over yet!'`** if there weren't enough moves provided to complete the game.

Notes:

- Boards will always be square. (Although, it might be interesting to see what tic-tac-toe games of different shapes would look like!)
- The only two possible players are **`'X'`** and **`'O'`**.
- As mentioned in the function background, if a move sends a player to a completed square, that player can choose any open board to play on. If that happens with the given sequence of moves, then the location of the next player's next move should be the first available board from the top left, going down the columns (in the same way as array indexing).
- If a move is played on a spot that is already taken, skip that move and follow the same procedure as above for choosing the location of the next move.
- There might be more moves in the input array than are necessary to complete the game. In these cases, you should stop making moves once the game is finished.
- If a board ties, fill it entirely with asterisks (**`*`**).

Hints:

- Don't be afraid of helper functions!
- If the calculations get too complicated, try removing the formatting (i.e. the border characters) from the board and then restoring it later.
- Think about writing a function to reduce an ultimate tic-tac-toe board to a regular tic-tac-toe board.
- **`find()`** has two outputs.

Function Name: `harmony`

Inputs:

1. *(double)* A list of chords

Outputs:

1. *(double)* A series of notes

Background:

This problem is all about music and the theory that goes into making music.

Chords, or permutations of notes played together, are one of the building blocks of music. A chord consisting of three notes, each a fixed distance in pitch apart from each other, is called a triad. Certain chords sound good when played after different chords, and we will be using that fact in this problem to create music. These are called chord progressions.

In music, a voice is a series of notes played by one or more people. One of the foundational elements of music theory is how to make the different voices in a piece of music sound good together. The simplest form of this, taught in introductory music theory classes, is four-voice part writing. The goal of part-writing is to arrange the notes in a way that sounds pleasing. For the purposes of this problem, we will be exclusively using triads in a major key. The four voices will we use are the ones in a traditional choir; soprano, alto, tenor, and bass. Each voice will sing one note at a time.

Function Description:

Given a series of chords of length N in the input, your function should generate a 4xN array of notes representing the best possible way to arrange the notes in each chord. Each value in the input vector represents one chord. We will represent the notes that can be sung as the integers between 1 and 27. For each chord in the input, you can use the function **`chordGen()`** to obtain all the possible ways to arrange the notes in that chord.

Find all possible arrangements of notes for the first chord by calling **`chordGen()`**. Then, find all the arrangements of the next chord. For each arrangement of the first chord, find the best arrangement of the notes in the second chord that matches the individual arrangement of the first chord.

To do this for a particular arrangement of the first chord, throw out all combinations of the first and second chord that do not meet the following rules:

- If any row has a number equal to 7 mod 7, the next note should always be equal to 1 mod 7. The note equal to 7 mod 7 is called a leading tone and "leads" into the 1 mod 7 note, or the tonic.
- If a pair of notes in a column (aka a chord) are separated by 4 mod 7 or 7 mod 7, they cannot be separated by the same distance in the next column, unless they are on the

exact same notes. For example, an arrangement where one column contains 15 and 19 and the next contains 16 and 20 is invalid.
- Two consecutive notes in a row cannot differ by 6 or anything greater than 7. Valid differences are {0,1,2,3,4,5,7}.
- The fourth row of the first and last columns should have the root, modulo 7.

Then, calculate the score of all the remaining combinations of the first and second chord. The score of a column can be calculated by multiplying the difference in the first and fourth row and the sum of the jumps in each row between the current column and the previous column. Take the absolute value of all the jumps and distances before adding/multiplying.

For example, if my progression was **`[1,5]`**, then the notes generated could look like this.

```
notes =>
[22   19
 19   16
 15   14
 8    12  ]
```

Then the score for the second chord would be:

```
|12-19| * [ |(19-22)| + |(16-19)| + |(14-15)| + |(8-12)| ] = 77
```

For each possible arrangement of the first chord, append the second chord arrangement with the score closest to the mean score of all the possible arrangements of the second chord for that first chord. If there are ties, then store each pair separately. After this, you should have the roughly the same number of possibilities as you had after the first chord, just with an extra chord attached to each possibility.

Say there were 15 valid arrangements of the first chord, and 20 for the second chord. For the first of those 15 chords (#1), you determine which of the 20 chords fulfills the condition above, then concatenate that best chord onto #1. Then, you take the second of 15 chords (#2) and determine which of the 20 chords fulfills the condition above. If two of the 20 chords have scores the same distance from the mean, then you'd create a copy of chord #2, and concatenate the first of the chords you found onto the original and the second of the chords you found onto a copy. Repeat for all 15 arrangements of the first chord.

Continue this procedure with the third chord, considering the score of each possible arrangement of the third chord with respect to the second chord, and so on, until you reach the end. Once you have generated the full progression for each possible arrangement of the first chord, evaluate the cumulative score for all of them (add each column's score together), then output the progression with the score closest to the mean of all the cumulative scores.

In the previous example, let's assume that all 15 of the arrangements of the first chord had a unique best second chord, and the third chord had 12 valid arrangements. You would continue by determining which of those 12 fulfills the condition above for the first of the original arrangements (#1), then doing the same for #2, etc.

Notes:

- This strategy of grabbing whichever chord is "best" compared to just the previous chord is known as a greedy algorithm.
- A function **`playMusic()`** has been provided that can play the notes that your function outputs back to you. Use help for more information on how it works. Make sure your volume is on!
- Examples:
  - **`playMusic(harmony_soln([1,5,1]))`**
  - **`playMusic(harmony_soln([1,6,4,2,5,1], 1, 'Eb')`**
- The score for the first column is zero.
- You are guaranteed to have a valid input (i.e if a chord contains a leading tone, the next one will have a root)
- There will always be at least three chords.
- If there is no valid next chord for a given progression, then delete that progression. This might happen if there's two leading tones in one chord but only one root in the next chord.
- There will always be a unique solution.

Hints:

- The **`repmat()`**, **`repelem()`**, and **`reshape()`** functions may all come in handy.
- One of the tricky parts of this problem is how to store the arrangements of chords. You could have each set of four rows be an arrangement and have the first chord be the first column, etc. You can set each column to be an arrangement and have rows 1-4 be the first chord, rows 2-8 the second, etc. You could store each arrangement in a cell array, or maybe even store subsequent chords in the third dimension. Be creative!
- The number of possible progressions should not greatly increase between consecutive chords. Each potential arrangement of the first chord gets built upon.