Function Name: `filterData`

Inputs:

1. *(double)* A vector of your original data

Outputs:

1. *(double)* A vector of your filtered data with outliers replaced with the median value

Banned Functions:

`iqr(), prctile()`

Function Description:

With all the data you will be asked to work with at Tech, there will be times when things don't go as planned, and when they don't, you'll get outliers. You'll have cases where you know what the data *should* be, but some of the values are a bit off. But thanks to MATLAB you can now efficiently fudge your data! (do not actually do this)

Write a function called `filterData` that takes in a vector of your original data and outputs a new data vector, which replaces values that are statistically defined as outliers from the median value Q_2. The median is the value in a set of numbers such that half of the elements are less than this value and half are greater. You **should** use the `median()` function to calculate this value. An outlier is defined as any value that has a value greater than Q_3 + (1.5*IQR) or less than
Q_1 - (1.5*IQR), where:

Q_1 = first quartile (median of the smaller half of sorted data, excluding the median Q_2)
Q_2 = second quartile (median of entire data set)
Q_3 = third quartile (median of the larger half of sorted data, excluding the median Q_2)
IQR = interquartile range (IQR = $Q_3 - Q_1$)

Your new data vector must maintain the order of the original data, with any outlier values replaced by the median value.

For example, with an original data vector:

```
[1 3 6 4 5 2 7 40]
```

`filterData` would output:

```
[1 3 6 4 5 2 7 4.5]
```

because `40` was the outlier, and the median value is `4.5`

Notes:

- Round all values to the nearest hundredth.
- Only round the final vector, do not round intermediate calculations.
- We do not endorse fudging data.
- Do not use the `iqr()` or `prctile()` functions

Hints:

- The `median()` function will be helpful.

Function Name: `elephantCase`

Inputs:

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

Outputs:

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

Banned Functions:

`strrep()`

Function Description:

You may already know of the popular variable naming convention called camelCase, but you probably have not heard of elephantCase (because it hasn't been a thing until now). Here are the rules for elephantCase:

Given a string...

1. Remove all numbers and punctuation (spaces should remain unchanged).
2. Capitalize the first and last letter of every 3-letter word.
3. Capitalize every letter of the word 'elephant' the first time it occurs.
4. Make sure all other letters in the string are lowercase.
5. Place an exclamation point at the end of the string.

   For example, the input:

   ```
   'African Elephants are > 500 lbs'
   ```

   Would become:

   ```
   'african ELEPHANTs ArE   LbS!'
   ```

Notes:

- There will always be **at least** one space before each word, except the first word, which may have none.
- There **can** be leading spaces or trailing spaces.
- The total number of spaces in the input string should equal the total number of spaces in the output string.
- `'elephant'` can appear in the input string in any case.
- `'elephant'` may be a substring of another word. For example, `'elephantiasis'`, in which case the output word should be `'ELEPHANTiasis'`.
- `'elephant'` may appear 0 or more times in the string. Only the first instance (if any) should be capitalized.

Hints:

- The `diff()` function may be useful.

Function Name: `multTable`

Inputs:

1. *(double)* A partially correct M by N multiplication table in array form

Outputs:

1. *(double)* An unweighted grade of the multiplication table
2. *(double)* A weighted grade of the multiplication table

Function Description:

Multiplication tables are commonly used to test elementary school students' mathematical aptitude. They include arrays of numbers, each number being the product of its row number and column number. Here's an example of a perfectly completed table:

```
multiplication_table =
      1     2     3     4
      2     4     6     8
      3     6     9    12
```

Now your research lab is studying the effectiveness of using said tables and there are a lot to grade. You start out by grading the tables on how many entries are correct. However, anyone can get the first couple of rows/columns correct because multiplication of smaller numbers is easy, whereas multiplication of larger numbers is tough. Therefore, in addition to an unweighted grade that values each entry equally, you must calculate a weighted grade too: a student receives points for the sum of their correct entries out of the sum of possible correct entries. Here's an incorrectly completed table (incorrect entries highlighted in red):

```
multiplication_table =
      1     1     3     4
      2     4     6     6
      3     6     8    12
```

A participant in your study missed a 2, 8, and 9 so their grade would be (1 + 3 + 4 + 2 + 4 + 6 + 3 + 6 + 12) / (1 + 2 + 3 + 4 + 2 + 4 + 6 + 8 + 3 + 6 + 9 + 12) * 100. In words, the weighted average is the sum of all values that are correct divided by the sum of all values in the correct multiplication table. The unweighted grade would simply be 9 / 12 * 100. (The number of correct values divided by the total number of values).

Notes:

- The output value should be expressed as a percentage, from 0 to 100, inclusive.
- Round each grade to four decimal places.
- The multiplication tables will always start at 1.

Hints:

- `size()` and the colon operator (`:`) will be useful.
- Yes, there are ways to do this problem with iteration and/or linear algebra. However, there are also ways to do it using only concepts discussed thus far in class.
- Consider the following:

```
[1, 1, 1, 1;...        [1, 2, 3, 4;...
 2, 2, 2, 2;...   .*    1, 2, 3, 4;...
 3, 3, 3, 3;...         1, 2, 3, 4;...
 4, 4, 4, 4]            1, 2, 3, 4]
```

- Can you create the above arrays using just the colon operator and indexing (and maybe a transpose)? What happens when you perform element-wise multiplication?

Function Name: `scheduler`

Inputs:

1. *(logical)* A 48 x 7 logical array representing your availability
2. *(char)* A comma separated string containing event information

Outputs:

1. *(logical)* Your updated schedule (still 48 x 7 array) after taking into account events you can attend

Function Description:

After many times having frustration with a common class scheduling software's reliability (which shall remain anonymous) (*cough* courseoff *cough*) you decided to leverage the power of MATLAB to do some scheduling for you! Write a function that takes in your current schedule and time information for some events or classes you are interested in and returns your new availability.

The first input, representing your current week schedule, is a 48 x 7 logical array. Each column is a separate day, while the 48 rows correspond to the 48 half-hour blocks in your day. The week starts on Sunday and goes through Saturday. A `true` in this logical array represents a half hour block that you are free, while a `false` represents a time you are unavailable. The second input is a comma separated string that contains event information. The string format for an event is as follows:

```
'<event name>,<day of week>,<time of day>'
```

The event name is simply a string containing the event name. The day of week is a string containing the day(s) that the event takes place. It will not have the day of the week as the word, rather as a number (1 is Sunday, 2 is Monday, etc.). If there are multiple days, this part will be a vector of numbers representing the days that the event takes place. The time of day part contains the time range of the event. It will be formatted in military time (hours range from 0 to 23), the hours and minutes will be separated by a colon (`':'`), and the beginning and ending time will be separated by a single dash (`'-'`). A sample valid event could be represented by the following string:

```
'Calculus,[2 4 6],11:30-13:00'
```

The entire event input will be comprised of 2 of these single entries, all also separated by commas so the whole input will look like the following:

```
'<event name 1>,<day of week 1>,<time of day 1>,<event name 2>,<day of week 2>,<time of day 2>'
```

Take the information in this string and update the schedule array taking these 2 events into account and output an array formatted exactly like the input one with your new availability.

Notes:

- You will participate in every event you can.
- If you can only participate in part of an event, you will participate in whatever part you can.
- If any events overlap, assume you will participate in one of them (it is irrelevant which one you choose, just that you will no longer be free during that time block).
- There will always be 2 events.
- Times will not cross over a day.
- The times inputted will only be in increments of 30 minutes (will either end in `':00'` or `':30'`).

Hints:

- You will find the `strtok()` function very useful.
- Notice there are no spaces separating the parts of the second input, just commas.

Function Name: `puzzleBox`

Inputs:
1. (*char*) A jumbled character array
2. (*double*) The row shift vector
3. (*double*) The column shift vector

Outputs:
1. (*char*) The solved character array

Banned Functions:
`circshift()`

Function Description:
Have you ever had the urge to get a wooden Chinese Puzzle Box and spend your free time trying to solve it, but realize that you have no free time? Well have no fear, now you'll be forced to solve one! Write a function called `puzzleBox` that will take in a jumbled character array and two vectors (representing a row shift and a column shift) and produce a satisfying ASCII image. The last elements in the row and column shift vectors represent the number of shifts to make, while all other numbers in the vector represent which row/columns to shift. For example, if the row shift vector was `[3, 5, 12, 7]`, this would indicate that you should shift every element in row 3, 5 and 12 to the right by 7 columns (shifts should wrap around to column 1 if they go past the right edge of the array). Positive row shifts move elements to the right and positive column shifts move elements down. Negative shifts move in the opposite direction. Your code should be formatted such that the rows slide first, and then the columns.

For example, given the following inputs:
```
jumbledCharArr = ['abc';...
                  'def';...
                  'ghi']

rowShift = [1, 3, 2];

colShift = [2, -5];
```

You should first shift rows 1 and 3 by 2 elements to the right (wrapping elements around the end of each column). This procedure will produce the array:
```
afterRowShift = ['bca';...
                 'def';...
                'hig']
```
Then you should shift column 2 by 5 elements up (wrapping elements around the end of each row). Because the array is a 3x3, shifting by 5 in the vertical direction is equivalent to

shifting by 1 in the downward direction (**cough** `mod()` **cough**). Therefore, the final character array produced by these inputs is:

```
finalCharArr = ['bia';...
                'dcf';...
                'heg']
```

Notes:

- The shift value can be any integer.
- The index elements will always consist of valid row/column indices

Hints:

- You will find the `mod()` function very useful.
- Character arrays are just normal arrays in disguise.