Notice

You may notice that the O (output) in High Level File I/O is left out of this homework. This is because `xlswrite()` does not work on Macs. For your convenience, we do not require you to use this function, except on the ABCs. However, you will still need to know the semantics of this function for the tests. We have set up some of the functions such that the output is a cell array that could readily be written to an Excel file, we just don't require that you actually write any files.

Happy Coding,
~Homework Team

Function Name: `cellSearch`

Inputs:

1. (*cell*) A 1xN nested cell array
2. (*double*) A 1xN sequence of indices

Outputs:

1. (*variable type*) The value contained at the specified location in the cell array

Function Description:

Deeply nested cell arrays can be a pain to index, so you will eliminate that problem by writing a function to index cell arrays for you! The first input to the function will be the cell array and the second input will be a vector of indices. The indices progressively narrow down your search through the cell array. This process is best illustrated through an example:

```
    cellArray = {{[45, 23, 10], 'hello'}, {'how', {'are'}, true}, 89,
'you'}
        index = [2, 2, 1, 3];
```

The first 2 in the index vector indicates that you should index the second element of the cell array. So now you are searching just within the cell array contained at the second index of the top-level cell array which is:

```
cellArray = {'how', {'are'}, true}
```

From here, you progress to the second 2 in the index vector. This, again, indicates that you should index the second element of this cell array, leaving you with:

```
cellArray = {'are'}
```

The next index is 1 so you index the first element, leaving you with `'are'` (no longer contained in a cell). Then the last index, 3, indicates that you should index the third element of this string, which is `'e'`.

At this point you have reached the end of the index vector, so you return whatever value you have discovered, in this case the character `'e'`. However, it is possible to find a value of any type as you go through the indexing process which is why the output is listed as having a "variable type" above. This process is functionally identical to saying `cellArray{2}{2}{1}(3).`

As you are indexing the cell array, whenever you come across a cell, you should open it; i.e. use curly braces to index whenever you can. Only once you have found a data structure that is not a cell should you use regular parenthesis to index.

Notes:

- There can be any number of elements in the index vector, but they will never be invalid indices of the given cell array.

Function Name: `warrenBuffett`

Inputs:

1. (*char*) The name of an Excel file containing stock information

Outputs:

1. (*char*) The name of the stock to invest in
2. (*cell*) A cell array representing what the new spreadsheet should look like

Function Description:

It's time to drop out of Georgia Tech and go where the real money is: Wall Street. You have decided to pursue a career in trading stocks, and in order to help you decide which stocks to invest in, you want to use MATLAB, the only useful thing you learned in college. Given an Excel file that contains information on a group of prospective stocks, write a function called `warrenBuffett()` that modifies this information and predicts which single stock you should put all your money into*. Here is an example of a spreadsheet table:

Symbol	Name	Price	Change
NVDA	NVIDIA Corporation	101.46	0.97
PFE	Pfizer Inc.	34.26	0.20
X	United States Steel Corporation	37.01	-0.30
VZ	Verizon Communications Inc.	50.60	0.29
XOM	Exxon Mobil Corporation	81.08	-0.70
TSLA	Tesla, Inc.	257	1.01

The spreadsheet will always have at least four columns with subject headers `Symbol`, `Name`, `Price`, and `Change`. But, there can be any number of additional columns and the columns can be in any order. Your function must do three things:

1. Move the `Symbol` column to the far left so that it is the first column in the table.
2. Calculate the percent change in price of each stock, which is

   ```
   Change/Price * 100
   ```

   Add a new column on the very right called `'% Change'` and put these values underneath. You should round your percent change values to the second decimal place.
3. Sort the table according to `% Change`, in descending order so that the stock with the highest rate of increase in price is at the top.

After you have performed these operations, output an edited cell array that represents what the new spreadsheet would look like.

Lastly, you should output a string of the name of the stock with the highest `% Change`. This is the name of the stock found under the `Name` column, NOT the `Symbol`.

Notes:

- You should keep the table headers in the first row in your outputted cell array.
- Round your percent change values to the second decimal place.
- Besides moving the `Symbol` column to the left and adding the `% Change` column, you do not need to reorder the columns.
- You do not need iteration to solve this problem.
- *Investing all of your money in one stock is a bad idea in the real world.

Hints:

- String functions work on a cell arrays of chars!

Function Name: `wordCounter`

Inputs:

1. *(char)* The filename of the text file that you want to count the words in

Outputs:

1. *(cell)* A 2xN cell array containing every unique word in the first row and the number of times it appears in the text in the second row

Function Description:

Have you ever wondered how many times a word has been said in a book? Or wanted to see if you kept using the same adjectives in your english paper? Or perhaps wanted to know how many times the person who presented in class today had the word "like" in their paper since they must have said it at least 50 times? With the powerhouse that is MATLAB, you now can!

You will be given a `.txt` file, and must find a way to extract every unique word and store them in a cell array, with the first row of the cell array containing the word, and the second row containing the number of times that word appears in the text. In other words, each column should have a word with the corresponding number of appearances in the text. Case should be ignored when looking for unique words, and any characters that are not letters or spaces should be removed. The words should appear in alphabetical order in the output cell array.

For example:

```
Twinkle.txt → 'twinkle 1738 twinKLe## little STAR'

out =
      {'little', 'star', 'twinkle'
          1,       1,       2    }
```

Notes:

- The words in the output cell array should be lowercase.
- If you want to test different files, or investigate a specific piece of work, check out archive.org and search for your text of choice.

Hints:

- It may be useful to initialize the cell array, among other items, before you enter a loop.
- Remember that many functions that work with strings will also work on cells containing strings. In fact, some functions that don't work on strings will work on a cell array containing strings *cough* `sort()` *cough*
- Cell arrays are just like normal arrays in the sense that you can delete and add/expand just like with normal arrays.

Function Name: `int2word`

Inputs:

1. *(double)* An integer between -999 and 999, inclusive

Outputs:

1. *(char)* A string representation of the integer

Functions Description:

Numbers are great, but sometimes it's necessary to write them out in text. However, this can be a tedious process, which is why you are going to automate it in MATLAB.

Given an integer, convert it to the fully written-out word form. For example, the integers:

`0`, `1`, `2`, `11`, `16`, `20`, `45`, `-100`, and `549`

correspond to

`'zero'`, `'one'`, `'two'`, `'eleven'`, `'sixteen'`, `'twenty'`, `'forty-five'`, `'negative one hundred'`, and `'five hundred and forty-nine'`,

respectively. The full set of rules for how to format numbers is listed below.

Rule Number	Value Range	Rule
(1)	`-999 <= x < 0`	Prepend `'negative '` to the string generated from `abs(x)`
(2)	`x == 0`	`'zero'`
(3)	`1 <= x <= 19`	Spell out the number (`'one'`, `'two'`, … , `'nineteen'`)
(4)	`20 <= x <= 99`	`'<prefix>-<`**(3)** applied to ones digit`>'` <prefix> is `'twenty'`, `'thirty'`, … , `'ninety'`
(5)	`100 <= x <= 999`	`'<`**(3)** applied to hundreds digit`> hundred [and] <`**(4) OR (3)** applied to remaining digits`>'`

These rules may look sort of confusing, but it's basically what you "expect" the answer should be. If you are unsure about the formatting, you can always run the solution function.

Notes:

- It is possible to do this problem without cells, but there is a reason this problem is on the cells homework.
- It is pretty easy to make a test script that can test your function against the solution for all possible inputs.
- The word `'and'` should only be present after the `hundred` part of the final phrase if there are more nonzero numbers following that place.

Function Name: `marchMadness`

Inputs:

1. (*char*) The filename of the Excel file containing points per game data
2. (*char*) The filename of the Excel file containing opponent points per game data
3. (*char*) The filename of the Excel file containing winning percentage data
4. (*char*) The filename of the Excel file containing seed data
5. (*char*) The filename of the Excel file containing what matches must be played

Outputs:

1. (*cell*) The completed 64x7 cell array representing the final bracket for March Madness

Function Description:

Late one night, still reeling over the Indians blowing a--- Thunder blowing a--- Warriors blowing a 3-1--- Falcons blowing a 25 point lead, you decide you will never support a losing team again! With March Madness fast approaching, you realize that your newly acquired MATLAB skills can help you make the perfect bracket. Instead of relying on biased speculations, your formula will use various statistics about the teams' seasons to determine the winner!

You are given several Excel files containing data relevant to your calculations. The files for points per game (PPG), opponent points per game (OPPG), and winning percentage (WPCT) will contain the name of the team in the first column and the value of the statistic in the second column. The seed (SEED) data file contains the seed in the first column and the name of the team in the second. The possible seed values are 1-16, meaning there will always be 64 total seeds in the file. The seed data has four teams for each seed because there are four divisions. They are presented in order, so the 4th seed for the 3rd division would be the third team with a 4 next to its name. Using this data, calculate the score for each team listed in the seed data file using the following formula:

$$Score = (\frac{1}{6} * \frac{PPG - OPPG}{PPG}) + (\frac{1}{6} * WPCT) + (\frac{2}{3}) * \frac{(16 - SEED)}{15}$$

After you've done this, use the final input, the matches file, to start your bracket. It contains a single column of the games (represented as seeds) you will need to simulate in the first round. You'll notice there are repeats again, and they follow the same pattern as the seed: the first 1 you encounter represents the 1st seed in the 1st division, the second 1 represents the 2nd division, and so on. Games should be read in pairs, i.e. the first two rows tell you the 1st and 16th seed in the first division should play a game against each other. After you've simulated the first round, use the winners to simulate the next round, using the first two winners for the first game, second two for the second game, etc. Continue simulating rounds until only one team remains.

In order to "play a game" and determine the winner of a matchup, we have provided a helper function `playGame(team1Score, team2Score)` that takes the scores of two teams as

inputs and returns a logical value indicating whether the first team you input won or not. DO NOT simply pick the highest score as your winner, and DO NOT submit the helper function with your homework.

Your final output should be a cell array in bracket-like format containing the names of the teams that won each round in their own column. They should be evenly spaced vertically, and empty arrays should be used to fill the blank spaces in the output bracket.
A visualization using a small portion of a possible input:

```
seed = {1, 'UGA'                                    matches = {1
        1, 'Wayne State'                                       2
        1, 'Mary Mac'                                          1
        1, 'Miami'                                             2
        2, 'Notre Dame'                                        1
        2, 'Chattanooga'                                       2
        2, 'Michigan State'                                    1
        2, 'Coastal Carolina'}                                 2}
```

The final bracket would look something like:

```
brakcet = {'UGA',               [],          [],          []
           'Notre Dame',        game1winner, [],          []
           'Wayne State',       [],          game5winner,  []
           'Chattanooga',       game2winner, [],          []
           'Mary Mac',          [],          [],          game7winner
           'Michigan State',    game3winner, [],           []
           'Miami',             [],          game6winner,  []
           'Coastal Carolina',  game4winner,  [],         []          }
```

Notes:

- See the solution file output for the specific format of the final bracket.
- Do not perform any rounding on the scores you calculate in the first part.
- A cell containing an empty array is different than an empty cell. Make sure your bracket is padded with the former.
- Not all of the teams with PPG data will play in the tournament.

Hints:

- Open up the excel files before you start to get a better idea about what they look like.
- You may find the `cell()` function useful when padding out the bracket.
- You will likely get a fraction when calculating the number of cells that should go above and below an element to space it evenly. `ceil()` that value to get the top padding and `floor()` it to get the bottom padding. For example, if you calculate that 1.5 cells need to go above and below to space the team's name evenly, put 2 empty cells above and 1 empty cell below the team's name.