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 inex 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:

- `iscell()` will be helpful
- There can be any number of elements in the index vector, but they will never be invalid indices of the given cell array.
- The cell array can contain any number of nested cells or any other data type we have learned about (char, logical, or double).

Function Name: `fantasyTanks`

Inputs:

1. (char) A string which contains the filename of a list of professional players.
2. (char) A string which contains the filename of your friends' picks for the week.

Outputs:

1. (char) A string which says who won by how many points this week.

Function description:

It's time to play the best fantasy pick-em out there: Fantasy Tanks! You're responsible for organizing this season's fantasy tanks for all your best tanking buddies. (This is a real thing that really exists).

This function will take in the name of two different files. The first file contains a list of players in the first column and points in the second. The two columns will always have the headers 'Player' and 'Points' respectively. The second file contains your friends names in the first column and the given friend's' player picks for the week in the remaining columns. Given these files, output a string which describes who won first place, who was in second place, and the difference in points between first and second place. A friend's score for the week is the sum of all the scores of the players they picked. For example, if the following scores were the results for the week:

Player	**Points**
canadianimpact	3500
Overlord_Prime	2750
OxThief	3300

And the following was a list of picks:

Name	**Picks**	
Nick	Overlord_Prime	OxThief
Emily	Ox_Thief	canadianimpact

Because Nick picked Overlord_Prime and OxThief, his total score would be:

2750 + 3300 = 6050

Emily's total score would be:

3300 + 3500 = 6800

The output should look like:

```
'<winner name> won by <point difference> points over <runner-up name>!'
```

If there is a tie for the winner, output the string:

`'It's a tie!'`.

In this case, Emily's score was higher, so the output would be:

`'Emily won by 750 points over Nick!'`

Notes:

- Each friend will pick the same number of players.
- The number of picked players will never be zero.
- Each friend cannot pick the same player twice, but the same player may be picked by multiple people.
- If there is a single winner, there will **not** be a tie for second place.

Hints:

- The first test case is designed to be small enough that you can easily step through your function and see exactly what is happening at each step.

Function Name: `careerFair`

Inputs:

1. *(char)* The name of an Excel (.xlsx) file containing data of applicants

Outputs:

1. *(cell)* An Nx2 cell array

Function Description:

Your company decides to send you to the career fair at Tech as a recruiter. You are supposed to talk to a bunch of applicants, tell them to apply online, and collect data of some good candidates. Being a good recruiter, you want to organize the data you collected before giving it back to HR.

You will be given the name of an Excel file containing data of all the candidates. The input Excel file will always contain columns labeled `'Name'`, `'Age'`, `'Year'`, and `'Resume'`, but not necessarily in that order. The Resume column will contain the name of a text file which contains the applicant's resume. Each resume is guaranteed to have a GPA on the first line of the file. The format of the GPA will always be: `'GPA: #.##'`. This substring will occur once, and only once, in the first line of the file. All GPAs will always show all three digits, even if leading or trailing digits are zero. For example, a GPA of .4 will be displayed as `'0.40'`

After finding the GPA of all candidates, you must output a cell array of the names of the candidates in order of decreasing GPA. The first column of the cell array should be the names and the second column should be the corresponding GPAs.

Function Name: `packIt`

Inputs:

1. *(char)* The name of a text file (.txt) containing ingredients and their calorie counts
2. *(char)* The name of an Excel (.xlsx) file containing recipes and their required ingredients

Outputs:

1. *(char)* A statement about what and how many calories you will eat for lunch

Function Description:

National Pack Your Lunch Day is coming up (March 10) and you've decided to celebrate by choosing one of your favorite recipes to take to class. The ingredients will always appear in the form:

`<amount>:<unit of measurement>:<item>`

Example Recipe Excel Spreadsheet:

`Tuna Sandwich`	`Tomato Soup and Grilled Cheese`	`Chicken Pesto Pasta`
`2:slice:bread`	`2:slice:bread`	`2:cup:pasta`
`.5:cup:tuna spread`	`2:slice:cheese`	`.5:lbs:chicken`
`.25:whole:tomato`	`2:whole:tomato`	`.2:whole:lemon`
	`1:tablespoon:cream`	`1:cup:pesto`

You are also currently trying to "get swole" and will therefore choose the recipe with the most calories. To compute the number of calories, you will be given a text file where each line is of the form:

`<item>:<calories>`

You can assume the units used in the recipe are the ones used in the calorie list (you do not have to do any unit conversion). You can also assume that all items in the Excel file will exactly match items found in the text file. Your answer will be returned as a string formatted as:

`'I will make <recipe name> for lunch and consume <num> calories.'`

`<num>` should be rounded to the nearest integer.

For more information about Pack Your Lunch Day, click here.

Function Name: `election`

Inputs:

1. *(char)* The name of an Excel (.xlsx) file containing candidate names and policy positions
2. *(cell)* A 15x3 cell array containing the nationwide data on voters

Outputs:

1. *(char)* A string declaring which candidate wins the election

Function Description:

It's election season in the nation of Shoelandia, and as the intense competition rages on, voters have begun making up their minds about important topics. As a political analyst, you've gathered every candidate's positions on the hottest topics and you are now trying to figure out which candidate has the best chance of winning the upcoming election. You are given an Excel sheet containing the names of candidates and their respective positions (yes/no) on ten issues, which you have labeled Policy 1 through 10. You are also given a nested cell array containing the voter information for the fifteen states of Shoelandia.

Format of candidate information Excel file (Nx11):

- The first row will be the headers: `'Candidates'`, `'Policy 1'`, `'Policy 2'`... (always in that order)
- All cells under the policy headers will contain a `'yes'` or `'no'` representing that politician's stance on that issue.

Format of voter information outer cell array (15x3):

- The first column will contain the names of the 15 states of Shoelandia
- The second column will contain an inner cell array (see below)
- The third column specifies the number of electoral votes each state gets

Format of voter information inner cell array (Nx2):

- Each row represents a voter in that state
- The first column of each row contains a 1x2 vector of the two policy numbers that the the voter cares the most about, with the most important being in the first index
- The second column of each row contains a corresponding 1x2 logical vector indicating whether the voter supports (true) or opposes (false) each of the two policies he/she finds most important

In order to calculate the candidate with the highest chance of winning the election, you must compare the stances of candidates to the stances of all the voters. Shoelandian voters are both picky and indecisive, so here is how you should predict votes:

- If **one and only one** candidate holds the same stance as the voter on only their highest priority policy (i.e. the first listed policy), then the voter will vote for that candidate.
- If **one and only one** candidate holds the same stance as the voter on their highest **and** second highest priority policy, then the voter will vote for that candidate.
- If neither of these conditions are met, the voter will not vote.

Here is an example:

Candidates Excel sheet (note that only 3 policies are shown for brevity):

Candidate	**Policy 1**	**Policy 2**	**Policy 3**
Sally	yes	yes	yes
Bob	yes	yes	no
Bernice	no	no	yes

Voter cell array for a single state (this will be a 3x2 cell array with 1x2 vectors in each cell).

`[1, 3]`	`[true, true]`
`[1, 2]`	`[true, true]`
`[1, 3]`	`[false, false]`

The first voter will vote for Sally because their 2 most important policies match Sally's. The second voter will not vote because both Sally and Bob policies match with voter 2. Voter 3 will vote for Bernice because her first priority policy has only one matching candidate (so the second priority policy should not be considered).

As a general rule: a person will vote if you can determine a single candidate match based on their top two policy issues. If multiple candidates are a match, then that person will not vote.

These votes will comprise the **popular** vote.

In addition, Shoelandia uses an electoral college that makes voting even more complicated. The candidate that receives the most votes within a state gets all of that state's **electoral** votes, the number of which is represented in the third column of the outer cell array.

The candidate with the highest percentage of electoral votes will be the one predicted to win the election, no matter what their popular vote percentage is.

Your output string should read:

```
'<candidate> wins with <electoral percent>% of the electoral vote and
                <popular percent>% of the popular vote.'
```

You should round all percentages to the nearest tenth of a percent.

Notes:

- There will always be ten and only ten policies.
- Open up the test case files if you are confused about the structure of the files.

Hints:

- Don't forget everything you know about masking just because you can use iteration now!
- We have provided a small test cases that you can use for debugging.

The voters of Shoelandia have grown bored of regular shoes and are craving for creative change. Try running the solution code with the name of a candidate who has both the musical and political experience to save Shoelandia from its lack of dope footwear. (This candidate's life is dope and he makes dope music, according to himself).