**Problem**

Given two files containing a list of words and a square grid of letters respectively, search the grid and cross-reference it with the word list. Print out the common words to the screen in alphabetical order, one word per line.

**Possible Solution**

**Pre-Processing Stage**

Read in file names from user input.

This can be done with the input() function

**Computation Stage**

Open the files, and consider the word file first.

We know that the minimum word length we're searching for in the grid is 3 characters, so we can reduce the size of our word file.

This can be done by creating a list to store words of valid length, and then iterating over the file and adding all words of valid length to the word list.

Next, consider the grid of letters. The data looks like a grid, which likely means that we want to come up with a data structure which can accurately represent a grid. In Python, a 2D list seems like a good choice, though in C/Java you may want to use an array if you know the size beforehand.

To create a grid from the file, we need to iterate over the file. Iteration will typically grab a line at a time, which is fine for our purposes. However, the characters in the grid are whitespace seperated, and storing the whitespace in our grid would increase computation time and memory usage. So, consider the **split()** method. If we split each line on whitespace, we end up with a list of only letters. If we append this list to another list created outside the iteration, we would have a 2D list, in which the second dimension of the list represents a row of letters.

Fun fact: the second dimension storing a row of the grid means we've created a "row-major" ordered matrix. From this, you should be able to infer what a "column-major" matrix would look like.

Thought process:

**What do we want to accomplish?**

We want to search this grid for words in the word file, but the rules of a word search says that words can be found in the horizontal rows, vertical columns, and diagonals (from upper left corner to bottom right corner)

**Problem Reduction:**
**We can reduce our problem of solving the word search to three smaller problems: horizontal search, vertical search, and diagonal search.**

**Horizontal Search**

By choosing to represent our grid in row-major order, we've made the right choice here. Our horizontal search amounts to indexing the grid to retrieve a row, joining the list representing a row into a string, and then appending that string to a list or similar structure, in order to give us a list of strings representing each row. We will wait to cross-reference this against the word list until a later date.

**Vertical Search**

Searching the columns can be reduced into two problems:

1. Write a function to return a string representing a column at some offset of the grid. (done in class)

2. Call that function on all offsets valid for the grid, and append the results to a list. We will cross-reference later.

**Diagonal Search**

Searching the diagonals can also be reduced into two problems:

1. Write a function to return a single diagonal at some offset. For example, on a 4x4 grid:

f i s h
h i k e
x s e a
a b c d

get_diagonal(grid, offset=0) = 'f'
get_diagonal(grid, offset=1) = 'hi'
get_diagonal(grid, offset=2) = 'xis'
get_diagonal(grid, offset=3) = 'askh'
get_diagonal(grid, offset=4) = 'bee'
.
.
.

2. get the diagonals at all the offsets, and assemble into a list.

**Output**
1. Concatenate the horizontal, vertical, and diagonal lists.

2. Iterate over the word list you made, note that now you won't have to check word lengths here. If a word in your word list is also in one of the strings in the concatenated list, then add it to an output list.

3. Sort the output list and print each word in it out, one line at a time.