Step 1: Follow the link: https://www.nltk.org/data.html to download NLTK data.

Step 2: Create a symbolic link to the NLTK data for this class.

Go to your working directory:

```
cd [working_directory_path]
```

Create a symbolic link to NLTK data:

```
$ ln -s [path_of_NLTK_data] [working_directory_path]/nltk_data
```

Step 3: Download and copy the project files to your working directory under Proj1 folder.

```
$ cp -r [path_of_downloaded_file] [working_directory_path]
```

Navigate to the project folder, and list its contents:

```
$ cd [working_directory_path]/Proj1
$ ls
```

You should see the following files:

- A.py
- B.py
- correct_outputA.txt
- correct_outputB.txt
- input.txt
- squirrel_girl.txt

III. Part A: Warm up with Linux and Python Basics

In this part of the assignment, you will learn a few handy Linux commands and play with a Python script. But wait! You've already learned some handy Linux commands, just by setting up your environment. Let's review those, first:

cd	Change directory. For example, run cd ~/Documents.
cp	Copy. If you want to copy a whole folder, you need to use the r option.
grep	Search for text. You need to tell it what to search for and where; for instance, grep import A.py will show you all the times "import" appears in A.py. Use the r option to search a directory instead of just one file. A shortcut to the current directory is a single dot, so grep – r resources . searched all of the files in the current directory.
ls	List a directory's contents. You can run it without any arguments to list the contents of the current directory, or give it a path to list the contents of a different directory. For example, if you're in your home directory, running ls ~/Documents will list the files in your Proj1 folder.
mkdir	Make a new directory

This is only the tip of the Linux iceberg. If you'd like to learn more Linux commands, you can find plenty of Linux cheat sheets like this `http://files.fosswire.com/2007/08/fwunixref.pdf` online. If you're looking for more detail, nixCraft `http://www.cyberciti.biz/` is often helpful.

Time-saving tip:

Lots of commands use the names of folders or files as arguments. Rather than typing these out, you can use tab completion to make Linux finish your thought for you. Try this: cd to your home directory, then start typing cd Doc, and hit the tab key. It should automatically expand to cd Documents/, and if you hit tab again, it will fill in your pin!

Now, let's move on to our files. If you open the code of A.py, you can see what's happening: Lines 1 and 2 import some modules —- pre-written code. Line 4 defines a variable (in this case, it's a string) called greeting, and line 5 prints it to the screen. In line 7, we use the nltk module's word_tokenize method to make a list of all of the tokens–words or punctuation in our greeting string. (Python lists are kind of like arrays that automatically grow or shrink as needed. Like an array, you can iterate through it or index into it.) We then use the for loop in lines 9 and 10 to print each token on its own line. Try running on your hw0 directory:

```
python A.py
```

You should get the output:

```
Hello, world!
The tokens in the greeting are
Hello
,
world
!
```

That's some really boring output, isn't it? Let's make it more interesting by replacing "Hello, world!" with text the user will provide on the command line. **First, modify Line 4 of A.py so that the command**

```
echo "This is a different greeting" | python A.py
```

yields

```
This is a different greeting
```

```
The tokens in the greeting are
This
is
a
different
greeting
```

Hint: If you're having trouble, take a look at this StackExchange Q&A: http://stackoverflow.com/questions/11109859/pipe-output-from-shell-command-to-a-python-script

What just happened? You used a pipe (|) to use the output of the first command as input to the second. Your first command was an echo, which just outputs the same thing you type in.

What if you already have a file with the text you want to use as input to your script? For example, suppose your GSI had already typed up the Squirrel Girl[1] theme song for you, and you didn't want to retype the whole thing. One thing you can do is use cat <filename>, which outputs the text of the file:

```
cat squirrel_girl.txt
```

should print the Squirrel Girl lyrics to the screen. You can also run A.py on these lyrics by piping the output of your cat command into your python command, like so:

```
cat squirrel_girl.txt | python A.py
```

There are a lot of repeated words in this song, aren't there? I wonder if there are more repetitions of the word squirrel or girl. Let's find out! Add the following line to the end of A.py:

```
print "There were %d instances of the word 'squirrel' and %d
instances of the word 'girl.'" % (squirrel, girl)
```

Now, modify the for loop to keep a count of the number of times "squirrel" appears and the number of times "girl" appears. Make the count case insensitive.

- Hint #1: Make sure you're initializing both variables (squirrel and girl) to 0 before the loop!
- Hint #2: The old standby "++" that you may know from other languages won't work in Python, but you can still use "+= 1"
- Hint #3: https://docs.python.org/2/library/stdtypes.html#str.lower

Half your grade for this assignment will be based on whether A.py is producing the right output, so you'd probably like to confirm that the output is correct. We have given you the correct output in correct_outputA.txt. You can check that your output matches ours by saving yours to a file, then using the diff command:

```
cat squirrel_girl.txt | python A.py > outputA.txt
diff outputA.txt correct_outputA.txt
```

where the > operator in the first line writes the text that would normally be printed to stdout into the file outputA.txt instead. If everything is working, the second line should return with no output; this means there were no differences between the two files. If you accidentally replaced the first line of the song with "This line shouldn't be here." and then ran diff, you would see

[1] https://en.wikipedia.org/wiki/The_Unbeatable_Squirrel_Girl

```
1c1
< This line shouldn't be here.
---
> Squirrel Girl, Squirrel Girl!
```

instead. This would show you exactly where your output was wrong, which should help you pinpoint the problem in your code.

IV. Part B: Word Similarity

Word similarity can play an important role in information retrieval. For instance, suppose a user asked for information about birds, and your system had to decide which of the following passages to return:

Stoats (Mustela erminea) were introduced into New Zealand to control rabbits and hares, but are now a major threat to the native bird population. The natural range of the stoat is limited to parts of the Northern Hemisphere. Immediately prior to human settlement, New Zealand did not have any land-based mammals apart from bats, but Polynesian and European settlers introduced a wide variety of animals[2]

Or

Eagles tend to be large birds with long, broad wings and massive feet. Booted eagles have legs and feet feathered to the toes and build very large stick nests. Ospreys, a single species found worldwide that specializes in catching fish and builds large stick nests. Kites have long wings and relatively weak legs. They spend much of their time soaring. They will take live vertebrate prey, but mostly feed on insects or even carrion.[3]

Clearly, the second passage is more relevant, even though "bird" appears once in the first passage and "birds" appears once in the second. As humans, we know that words like "eagles" and "ospreys" are similar to "bird," while words like "stoats" and "rabbits" are not very similar to "bird."

In this part of the assignment, you will compare three word similarity measures to determine which agrees the most with human ratings of similarity. The three methods are Lin similarity, Resnik similarity, and a vector-based similarity measure. Lin similarity and Resnik similarity are both based on WordNet (`https://wordnet.princeton.edu/`), a hand-built taxonomy of English words. The vector-based method represents words with GloVe vectors that were automatically learned by a system that looked at a corpus of billions of words and measures how similar the words are by how close together their vectors are.[4] In the vector-based method, the cosine between vectors often represents closeness. You will also see that word vectorization is important for neural network based algorithms as well in

[2]From `https://en.wikipedia.org/wiki/Stoats_in_New_Zealand`
[3]From `https://en.wikipedia.org/wiki/Bird_of_prey`
[4]You can learn more about these methods in the following papers:

- Philip Resnik. Using information content to evaluate semantic similarity in a taxonomy. Proceedings of the 14th international joint conference on Artificial intelligence (IJCAI'95).
- Dekang Lin. An Information-Theoretic Definition of Similarity. In Proceedings of the Fifteenth International Conference on Machine Learning (ICML '98)
- Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation.

later lectures. Such word vectorization is also called word embedding in the literature since it embeds a discrete space with one dimension per word into a continuous space with lower dimension.

B1: Parse the input file.

First, implement parseFile. This function will be given the name of a file that contains a word pair and a similarity score on each line. To see an example of the format of the file, look at input.txt. Your function should read in the file and return an ordered dictionary of (word1, word2): score entries. Make sure that you represent your scores as floats.

Hint: You can find out more about file I/O in Python here: `https://docs.python.org/2/tutorial/inputoutput.html#reading-and-writing-files`

B2: Use WordNet to measure similarity

Next, implement linSimilarities and resSimilarities using NLTK. These functions are given a list of word pairs and a WordNet IC corpus, and each should return a dictionary of (word1, word2): score entries. For this exercise, represent each word in the pair as the first synset in the list of noun synsets related to the word. If no noun synset is found for a word in the pair, represent each word as the first synset in the list of verb synsets related to the word. Note that the NLTK Lin scores are between 0 and 1; these will need to be scaled by a factor of 10 to be comparable to the human similarity scores.

Hint: Use the wn.synsets() method.
Hint: You can find out more about NLKT's similarity functions here: `http://www.nltk.org/howto/wordnet.html`

B3: Use the vector-based similarity measure

Look at the main function. Notice the following lines that are commented out:

```
model = gensim.models.Word2Vec()
model = model.load_word2vec_format(RESOURCES+'glove_model.txt', binary=False)
```

These lines will load a pre-trained vector-based model into memory[5]. Uncomment them. Before you do anything else, try running your code again, and notice how the runtime changes.

Now, implement vecSimilarities. The usual list of word pairs and the pre-trained model will be passed in to your function. Like the other functions, it should return a dictionary of (word1, word2): score entries. You can use Gensim to get the cosine similarity between the vector for word1 and the vector for word2; you'll need to multiply by ten like you did with the WordNet functions. Note you may need to convert the word to lower case.

Hint: See what the model can do here: `https://radimrehurek.com/gensim/models/word2vec.html`

[5]Wondering why the method is called Word2Vec when we're actually using GloVe vectors? Word2Vec and GloVe produce extremely similar vectors, but GloVe is a little newer. Gensim doesn't officially support GloVe yet, but by inserting a header into the GloVe file, we trick Gensim into being able to use all the Word2Vec functions with our GloVe vectors

B4: Save your output and check your work

Use the same technique you saw in Part A to save the output of Part B to outputB.txt. We have provided correct_output_B.txt, so you can diff your output against ours.

This should be your final output:

- A.py
- B.py
- outputA.txt
- outputB.txt