I chose to implement the “New Words” solution by using a binary search to the compare the provided tuples. Because the wordlist was already sorted, it seemed clear that the binary search algorithm was a straightforward way to achieve O(nlogn) on this assignment. The reading of chapter 3 reinforced this idea.

If I were to improve my code I would try and find a work around to converting my list to a tuple. Creating a new list to return with the correct words, and then converting to a tuple doesn’t seem like an efficient option. I attempted to use tuples and splice to assign new elements for the tuple to be returned but ended up having errors with that implementation. Ultimately, I settled on the list to tuple conversion for simplicity of creating the return object. I’m still learning python and would be curious about a work around for this case in future.  
  
I did a better job separating my subroutines than my Lab 1 submission. I tried to keep them to the “a function does one thing rule”. I also renamed a few of them to be better suited to what they complete during execution.

I considered using the tip from zyBooks about the sorted() method, but struggled to come up with the benefits of comparing a sorted words with the sorted wordlist. To ensure that all words are compared, I’d have to iterate through all elements of words to compare to wordlist regardless. Thus, there would be extra computation added without a benefit I could see at the time.

The big-O complexity of this algorithm is O(nlogn). O(n) because I iterate through each element of words but only iterate through log n of elements in wordlist.