**Project 3**

**List of notable obstacles:**

1. At first, I struggled with the permutation function. In every run of the function, I only caught half of the permutations. I later fixed the function by resetting rest and prefix after each recursive function call. In short, I was originally skipping some permutations as I called the function recursively *after* updating rest and prefix. By creating temp variables before the call and resetting them afterwards I could catch all the permutations.
2. I had difficulty in initially implementing the loop as a recursive function. However, I was rewatching the lecture and heard the professor mention the fact that we may have to implement co-recursive functions in project 3. When I implemented my storePermuation and Loop functions recursively, I was able to create a helper function that could directly be called by theJumbler.
3. I found the testing component of this project the most difficult and I’m still not 100% confident in the ability of my program. I managed to test small words in small dictionaries very easily (in fact even small words in big dictionaries but there I could not know whether I’ve caught all the valid permutations.) However, with big words (such as “regardless” which took 53 minutes to run on my computer), I was very unsure of the best way to test it. I tried very hard to make my function more efficient, but this complicated the code and made me less certain of its accuracy at every step. I also tried to increase efficiency by first implementing merge sort to sort the dict, then binary search at every step to check if a permutation matched (or there was a duplicate). To my initial surprise, this didn’t make much of a difference. Then, I realized that this was because the time taken to search for a match was not a function of the input size but rather a function of the dict size which is constant here. However, finding permutations is a function of the input size and I was unable to find a way to make that more efficient. I finally took comfort in the fact that someone posted on piazza that regardless took an hour for them to run and the professor said that’s acceptable.

**Test cases**

In this project, I found that it was highly inefficient to keep creating example dictionaries and writing up long assert statements to test things that can be verified manually in a matter of seconds. Therefore, I have listed below (without the assert format) the different words that I tested on the dictionary provided (words.txt) and briefly explained the other types of test that I conducted.

*Test cases using words.txt*

**Please enter a string for an anagram:** cat

Matching word cat

Matching word act

Program ended with exit code:

**Please enter a string for an anagram:** aeprs

Matching word parse

Matching word spare

Matching word spear

Program ended with exit code: 0**: 0**

**Please enter a string for an anagram:** aelst

Matching word least

Matching word slate

Matching word stale

Matching word steal

Program ended with exit code: 0

**Please enter a string for an anagram:** opst

Matching word post

Matching word spot

Matching word stop

Program ended with exit code: 0

**Please enter a string for an anagram:** aegilnrt *(42 secs)*

Matching word integral

Matching word triangle

Program ended with exit code: 0

**Please enter a string for an anagram:** aekst

Matching word skate

Matching word stake

Matching word steak

Program ended with exit code: 0

No matches found for:

1. regardl (6 secs)
2. regardle (40 secs)
3. regardles (6 mins)
4. regardless (53 mins)

I also adjusted the sizes of MAXDICTWORDS and MAXRESULTS respectively, setting them to values smaller than they should be (including 0 and 1) to check whether they are being cutoff correctly without causing any bad access errros.

*Test cases* ***NOT*** *using words.txt*

To reduce any biases in my testing, I modified words.txt and saw the results. Here are some common themes I covered with my tests. I performed more tests than the ones listed, but these are enough to prove a picture:

1. Cut the dictionary in half and then retested the words listed above.
2. Added “atc”, “tac”, “tca” “, cta” to words.txt and then tested “cat” expecting to see all 6 permutations of the string.
3. I used other sources of words such as <https://gist.github.com/deekayen/4148741> and <https://www.mit.edu/~ecprice/wordlist.10000> to find more permutations and see if my program could handle larger dictionaries.
4. Made it an empty word file and no words matched (except the empty string which is the default construction for string)