Pseudocode for helper functions

i) is_vowel()

It accepts a character as a parameter and returns a bool value type. It has an if else conditional statement and checks if the character passed is a vowel or not. It does this by comparing to all the vowels using an “or” (||). Else it returns false.

ii) is_consonant()

It accepts a character as a parameter and returns a bool value. This also has an if-else statement. It compares it to vowels . And if the parameter matches with any vowel, then it returns false, else it gives true. So, this is basically the reverse of is_vowel().

iii) Ends_with()

It accepts 2 string (one as a candidate and the other as a string), and returns a bool value. The first if statement check if candidate and string are empty strings, and the corresponding else if checks if the candidate is empty and the suffix is not.Then another else-if statement checks if the suffic is empty and the candidate is not. In this case it returns true. We add another if statement to ensure that the candidate is bigger than the suffix, if not then it returns false. Now we create a variable called end_of_candidate and store the candidate’s suffix in that variable. We use the substr function to pick a chunk of characters from the candidate. The starting point is the different in the length of the candidate and that of suffix, and goes till the length of candidate -1.

iv) Ends_with_double_consonant()

It accepts a string and returns a bool value. Firstly it checks if the length of the parameter is greater than 2. If we don’t check this, then the index of second last char may go out of range.
It then saves the values of second last and the last char of the string in two different variables.

We do this by using the string.at() function. After that we use an and , to compare if both of them are consonants. We then use a nested if to check if they are equal or not.

v) Ends-with_cvc()

It accepts a string as a parameter and returns a bool type. We first use an if statement to check if the length of the parameter is more than 3. Because of it’ll be less than 3 , then it can cause an index out of range error. It uses the is_consonant and the is_vowel function for the same. The key point here is that the index of the string.at() function starts from 0, however the length function starts from 1. So we find the character which is third last from the end and pass it to a parameter to the is_consonant function. We do the same for vowel and consonant.

vi) Count_consonants_at_front()

It accepts a string as a parameter and returns an int, which tells us the number of consonants present in it. A range based for loop is used for this , which goes character by character in the given string, and stores the value of that character in c for each iteration. If that c is a consonant(checked after passing it to the is_consonant funciton), then it increments a variable called const_present. Else it breaks the loop

vii) Count_vowels_at_back()

It accepts a string as a parameter, and returns an int. This can be done by using a while loop, by running it backwards. I starts from string.length()-1. The while loop will run as long as I<=0 and there is a vowel for that corresponding i(is_vowel(string[i])). We can add a counter which increases , and the I decreases with every iteration.

viii) Contains_vowel()

It accepts a string as a parameter and returns a bool value. It is similar to the Count_consonants_at_front(), as this also uses a range based for loop and check if that character is a vowel or not by passing it as a parameter to the is_vowel() function.

ix) New_ending()

It accepts a string(candidate), an int(suffix_length), and a string(suffix), it then returns a new string. Firstly we make a new string and copy the root word from the candidate by using the substr function to the new_string. We then append the replacement to the new string .