**The University of Texas**
**Computer Science**

# 313E Programming Assignment 09
# Reducible Words

## Setup

This assignment may be completed individually or with a pair programming partner. Please review and abide by the pair programming guidelines! Be sure to include your and your partner's name and EID in the Python file header. If you are working alone, you may delete one of them.

Complete these steps to prepare for the Assignment.

| Check | Description |
|---|---|
| ▢ | Download reducible.py and words.txt. You will be working on the reducible.py file. |
| ▢ | Place both files in the same folder/directory. |
| ▢ | You may not change the file names. Otherwise, the grading script will not work. |

If you would like to turn on autosave in VSCode, click on **File -> Autosave.**

## Problem Description

What are some of the longest English words that remain valid English words as you remove one letter at a time from those words? For this assignment, you will be implementing the concept of reducible words. Imagine that letters can be removed anywhere from a word, one at a time, with the stipulation that you cannot rearrange the remaining letters to form a valid word. Every time you remove a letter, the remaining letters form a valid English word. Eventually you will end up with a single letter, and that single letter must also be a valid English word. A valid English word is one that is found in the Oxford English Dictionary or the Webster's Dictionary. For this assignment, we will call these words reducible words. Here are two examples of reducible words:

1: **Sprite**. If you remove the r you get "spite". Remove the e and you get "spit". Remove the s and you get "pit". Remove the p and you get "it". Remove the t and you get "i" or "I" which is a valid English word.

2: **String**. Take away the r and you have "sting". Take away the t and you have "sing". Take away the g and you have "sin". Take away the s and you have "in". Take away the n and you have "i" or "I" which is a valid English word.

All reducible words will reduce to one of three letters - a**, i**, and **o**. We will not accept any other letter as the final one letter word.

**Requirements**

For this programming assignment, you will find the longest list of words that are reducible from a word list file, **words.txt**. You will print each word in alphabetical order on a line by itself.

1. Your program uses the provided text file **words.txt** to create your final output. You will be using input redirection to read from this file.
2. You will not write code to read and create your word list from the input file. We have provided this code for you in **main()**.
3. **You may not use a builtin set, dictionary or frozenset.**
4. **You may not change the names or parameters of the functions listed. They must have the functionality as given in the specifications. You can always add more functions than those listed.**
5. **You may not import any additional external libraries in your solution besides sys.**

You will modify the following functions to implement the functionality below.

**def step_size (s, const)**

Using a string and a constant, use double hashing to return the step size for that string.

**def insert_word (s, hash_table)**

Using a string and a hash table, enter the string in the hash table and resolve collisions by double hashing.

**def find_word (s, hash_table)**

Using a string and a hash table, find whether the string is in the hash table. Return True if it is, return False otherwise.

**def is_reducible (s, hash_table, hash_memo)**

Using a string, a hash table, and a hash_memo, recursively find if the string is reducible. If the string is reducible, it is entered into the hash memo and returns True. Otherwise, it returns False.

**get_longest_words (string_list)**

Using a string list, find and return a list of words that have the maximum length.

**main()**

In this method, you will determine the prime number N that is greater than twice the length of the word_list, then create and populate an empty hash_list. Next, you will hash each word in word_list into hash_list, resolving collisions using double hashing. You will also create a hash_memo and populate it with strings. You can assume that 10 percent words will be reducible and that the size of the hash_memo is a prime number that is slightly greater than 0.2 * size of word_list. You will then create a list of reducible_words. For each word in the word_list, you will **recursively** determine if it is reducible. As you recursively remove one letter at a time, check first if the sub-word exists in the hash_memo. If it does, then the word is reducible and you do not have to test any further. You will add the word to the hash_memo.

Lastly, you will find the largest reducible words in reducible_words and print the reducible words in alphabetical order, one word per line.

**Double Hashing**

Double hashing is an open-addressing technique used in hash tables to resolve collisions. It involves using two different hash functions to calculate the index where an element should be stored in the hash table. The first hash function is used to compute the initial hash value, and the second hash function is used to determine the step size for the probing sequence.

Suppose we have a list of keys, **[12, 24, 34, 42]** and a hash table size of **5**. Let's walk through an example of us inserting these keys into the hash table.

1. The size of the hash table is 5. We will use this value to calculate the index of where our value should be inserted by completing a mod operation, **index = key_value % table_size**. 12 % 5 = 2, so we will insert our first value **12** at index 2 of our hash table.
   ➔ **[None, None, 12, None, None]**
2. Next, we will try to insert key value **24**. 24 % 5 = 4. There is no key at index 4, so we successfully insert it into the hash table.
   ➔ **[None, None, 12, None, 24]**
3. Next, we will try to insert the key value **34**. 34 % 5 = 4. There is already a key at index 4, meaning that a collision has occurred. In order to solve this collision, we will use our double hashing technique. We already have our initial hash value, **4.** Next, we need to call another function to calculate a step size for the probing sequence.
   a. In order to use double hashing, we must define a constant prime number to use. It is common practice to use the next smallest prime number closest to the size of the hash table. In our case, the closest prime number to 5 would be **3.**
   b. Using this constant number, the step size method called will perform the operation using the formula: **step_size = constant - (key_value % constant)**
      i. 3 − (34 % 3) = **3**
   c. Using this step size, we will probe the hash table by adding 3 to our initial hash value, and wrapping back over to the beginning when we exceed the size of the hash table. To calculate the new index, we can use the formula: **next_index = (current_index + step_size) % table_size**
      i. (4 + 3) % 5 = 2. There is already a value at index 2, so we will continue the probing sequence.
      ii. (2 + 3) % 5 = 0. There is no value at index 0, so we can successfully add our key value **34** to our hash table at index 0.

   ➔ **[34, None, 12, None, 24]**
4. Lastly, we will try to insert the key value 42. 34 % 5 = **2**. There is already a key at index 2, meaning that a collision has occurred. We will use our double hashing technique again.
   a. Using our constant prime number **3**, the step size method will perform the operation using the formula: **step_size = constant - (key_value % constant)**
      i. 3 − (42 % 3) = **3**
   b. Using step size **3**, we will calculate the new index to begin the probing sequence.
      i. (2 + 3) % 5 = 0. There is already a value at index 2, so we will continue the probing sequence.
      ii. (0 + 3) % 5 = 3. There is no value at index 3, so we can successfully add our key value **42** to our hash table at index 3.
5. There are no more keys in our list, so our final hash table looks like this: **[34, None, 12, 42, 24]**.

To calculate the step size for our program, you can pick a prime number smaller than the size of the table for your own testing. You should expect your code to work for any prime number smaller than the size of the table.

**Memoization**

Memoization is a technique used typically with recursive algorithms to optimize and improve functions by caching results from previous function calls. Results are stored, usually in a dictionary, so that future calls to the same function with the same inputs can be retrieved from the dictionary cache instead of being recalculated again. Below is a step by step explanation of the process of memoization:

1. A function is called with an input. The memoization technique checks if the result from that input has already been stored in the cache, also known as a memoization table.

2. If the result is found in the cache, the result is returned immediately without performing any additional calculations or operations.
3. If the result is not found in the cache, the result is calculated and then stored in the cache for future use.

**hash_word(s, size)**
This method is used to compute the hash value of a given string “s” given the size of a hash table. It converts each character to its corresponding numerical ASCII value using the ord() method, and multiplies the hash index by 26 (for 26 letters in the alphabet), and taking the module of the hash index with the “size” to ensure that the new hash index fits within the size of the hash table. In short, this method converts a string into a numerical index for a hash table of a specific size.

**Input**
Your program uses the provided text file words.txt as input. You will be using input redirection to read from this file. All the words are in lower case and are two letters or more in length. The code can assume that the file is structured properly.

You can run the program by typing the following command into your terminal:

```
python3 reducible.py < words.txt
```

Alternatively, if you are using a Windows computer and are having trouble with input redirection <, you can run:

```
cat words.txt | python3 reducible.py
```

**Output**

**Below is the output of a successful run using the words.txt dictionary.**

```
complecting
```

For this programming assignment, rather than testing your output, our test cases will test individual functions for correct functionality.

## Grading

**Visible Test Cases - 75/100 points**

The program output exactly matches the expected output for each function. To receive full credit, the program must produce the correct output based on all requirements in this document and pass all the test cases.

**Hidden Test Cases - 15/100 points**

The program output exactly matches the expected output for each function. To receive full credit, the program must produce the correct output based on all requirements in this document and pass all the test cases.

**Style Grading - 10/100 points**

The reducible.py file must comply with the PEP 8 Style Guide.

If you are confused about how to comply with the style guide, paste the error or the error ID (e.g. C0116 for missing function or method docstring) in the search bar here in the Pylint Documentation, and you should find

The TAs will complete a manual code review for each assignment to confirm that you have followed the requirements and directions on this document. Deductions will occur on each test case that fails to follow requirements

## Submission

Follow these steps for submission.

| Check | Description |
|---|---|
| ▢ | Verify that you have no debugging statements left in your code. |
| ▢ | Submit **only** reducible.py (the starter code file with your updates) to the given assignment in Gradescope. This will cause the grading scripts to run. |
| ▢ | If you have a partner, make sure you are submitting it for you **and** your partner on Gradescope. See this Gradescope documentation for more details. |
| ▢ | When the grading scripts are complete, check the results. If there are errors, evaluate the script feedback, work on the fix, test the fix, and then re-submit the file to Gradescope. You can submit as many times as necessary before the due date. NOTE: By default, only the most recent submission will be considered for grading. If you want to use a previous submission for your final grade, you must activate it from your submission history before the due date. |

## Academic Integrity

Please review the Academic Integrity section of the syllabus. We will be using plagiarism checkers, and we will cross-reference your solution with AI-generated solutions to check for similarities. Remember the goal in this class is not to write the perfect solution; we already have many of those! The goal is to learn how to problem solve, so:

- Don't hesitate to ask for guidance from the instructional staff.
- Discuss with peers about **only** bugs, their potential fixes, **high-level** design decisions, and project clarifying questions. Do not look at, verbalize, or copy another student's code when doing so.

## Attribution

Thanks to Dr. Carol Ramsey and Dr. Shyamal Mitra for the instructions template and this assignment.