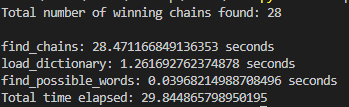
**Version 0 Code Performance:**

2-Word Solutions:



2-word and 3-Word Solutions:

5+ Minutes Allowed, no response, therefore terminated.

**V1: Early Termination**

# Base case: If all letters have been used, stop exploration, and add chain to valid\_chain\_solutions

if len(remaining\_letters) <= 0:

valid\_chain\_solutions.append(start\_chain)

return

# Find the last word and its last letter in the start\_chain

last\_word = start\_chain[-1]

last\_letter = last\_word[-1]

# Select every word as a potential continuation of the chain

for word in possible\_words:

if word[0] == last\_letter:

new\_start\_chain = start\_chain + [word]

if len(new\_start\_chain) > 2: # E

continue

new\_remaining\_letters = remaining\_letters.copy()

for letter in word:

if letter in new\_remaining\_letters:

new\_remaining\_letters.remove(letter)

# Recursively call generate\_chain with the updated start\_chain and remaining\_letters

generate\_chain(new\_start\_chain, new\_remaining\_letters)

In the code above, the highlighted if statement determines if the chain has reached a length of 3 words (2-word solution) and terminates the search. Prior to this condition check, many assignments are made that prove to be unneeded. We can instead check the length of the chain upon entering the generate\_chain function, as a second base case. (Change made below)

# Base case: If all letters have been used, stop exploration, and add chain to valid\_chain\_solutions

if len(remaining\_letters) <= 0:

valid\_chain\_solutions.append(start\_chain)

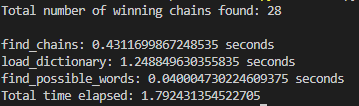
return

if len(start\_chain) > 1:

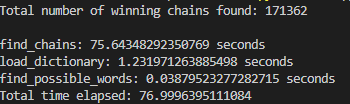
return

Let’s see how the performance has changed.

2-Word Sols:



2-word and 3-Word Sols:



Wow. As we can see the early termination to avoid the many initializations significantly reduced the run time. Let’s see what else we can do.

**V2: Load Dictionary**

The load\_dictionary function is used to insert all of the words from the dictionary.txt file for efficient searching.

def load\_dictionary(file\_path):

trie = Trie()

with open(file\_path, 'r') as file:

for word in file:

trie.insert(word.strip().upper())

return trie

However, each time the dictionary is loaded with the entirety of the words provided, and this is simply not needed for this use case. Let’s adapt the load\_dictionary function to create a trie of words consisting only of the provided game/puzzle letters.

def load\_dictionary(file\_path, gameData):

trie = Trie()

with open(file\_path, 'r') as file:

for word in file:

word = word.strip().upper()

# Flatten the list of letters from gameData

puzzle\_letters = [letter for letters in gameData.values() for letter in letters]

# Check if the word can be formed using the letters from gameData

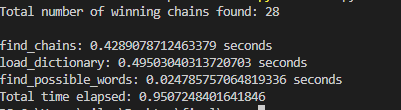
if all(letter in puzzle\_letters for letter in word):

trie.insert(word)

return trie

Since the number of puzzle letter remains the same, for testing we are only going to test 2-word solutions to determine the performance increase of the load dictionary function.

2-Word Sols.:



We do see a performance increase for both the load\_dictionary function and find\_possible\_words. While not huge, it definitely helps.