Python Code for Solving the Padlock Problem Using Combination Generation

Problem Recap

Farmer John wants to find the best five-letter word to use as the third row on his padlock that will maximize the number of valid word combinations. The padlock consists of three rows, each with five letter discs that can rotate to form different combinations.

• Fixed Rows: “BELLA” and “LOVES”

• Flexible Row: A word of choice to maximize valid words

The goal is to find the third row word that allows the most valid words to be created when each disc is rotated.

In [1]:
```python
# Load valid words from a file into a list
def load_valid_words(filename):
    valid_words = []
    with open(filename, 'r') as file:
        for line in file:
            valid_words.append(line.strip().upper())
    return valid_words

# Generate possible combinations by rotating letters in each slot
def generate_combinations(fixed_row1, fixed_row2, flexible_row):
    all_combinations = []
    for i in range(5):
        for j in range(5):
            for k in range(5):
                for m in range(5):
                    for n in range(5):
                        word = fixed_row1[i] + fixed_row2[j] + flexible_row[k] + fi
                        all_combinations.append(word)
    return all_combinations

# Check if a combination is in the list of valid words
def count_valid_words(combinations, valid_words):
    count = 0
    for word in combinations:
        if word in valid_words:
            count += 1
    return count

# Find the best third row word
def find_best_word(valid_words, fixed_row1="BELLA", fixed_row2="LOVES"):
    max_valid_combinations = 0
    best_word = None

    for candidate_word in valid_words:
        # Generate all combinations for the candidate word
        combinations = generate_combinations(fixed_row1, fixed_row2, candidate_word
```

```
        # Count how many of these combinations are in the valid words list
        valid_count = count_valid_words(combinations, valid_words)

        # Check if this candidate has the most valid combinations
        if valid_count > max_valid_combinations:
            max_valid_combinations = valid_count
            best_word = candidate_word

    return best_word, max_valid_combinations

# Main function
def main():
    # Load valid words from file (assumes valid_words.txt in the same directory)
    valid_words = set(load_valid_words(r'C:\Users\s563104\OneDrive\Personal\valid_w

    # Find the best third row word
    best_word, max_valid_combinations = find_best_word(valid_words)

    print("Best word for the third row:", best_word)
    print("Maximum valid combinations:", max_valid_combinations)

# Run the main function
if __name__ == "__main__":
    main()
```

```
Best word for the third row: VIGIL
Maximum valid combinations: 29
```

Determine the best N words

Provided an integer, N, determine the best N words which can be used as a third word in the combination lock for generating the maximum number of valid words.

In [2]:

```
from itertools import product

# Load valid words from a file into a set for quick lookup
def load_valid_words(filename):
    with open(filename, 'r') as file:
        valid_words = set(line.strip().upper() for line in file)
    return valid_words

# Generate all possible combinations by rotating letters in each slot
def generate_combinations(fixed_row1, fixed_row2, flexible_row):
    all_combinations = set()
    for letters in product(*zip(fixed_row1, fixed_row2, flexible_row)):
        word = ''.join(letters)
        all_combinations.add(word)
    return all_combinations

# Find the best N words that maximize valid combinations
def find_best_n_words(valid_words, fixed_row1="BELLA", fixed_row2="LOVES", N=1):
    word_combinations = []
```

```
    for candidate_word in valid_words:
        # Generate all combinations for the candidate word as the flexible row
        combinations = generate_combinations(fixed_row1, fixed_row2, candidate_word

        # Count how many of these combinations are in the valid words list
        valid_count = sum(1 for word in combinations if word in valid_words)

        # Append the candidate word and its valid word count to the list
        word_combinations.append((candidate_word, valid_count))

    # Sort words by the valid combination count in descending order and select top
    word_combinations.sort(key=lambda x: x[1], reverse=True)
    return word_combinations[:N]

# Main function to run the code for the best N words
def main():
    # Load valid words from file
    valid_words = load_valid_words(r'C:\Users\s563104\OneDrive\Personal\valid_words

    # Find the top N best words
    N = 5  # Example: Find the top 5 words
    best_words = find_best_n_words(valid_words, N=N)

    print(f"Top {N} words for the third row:")
    for word, count in best_words:
        print(f"{word}: {count} valid combinations")

# Run the main function
if __name__ == "__main__":
    main()
```

```
Top 5 words for the third row:
CANDY: 30 valid combinations
PASTY: 29 valid combinations
DUSTY: 29 valid combinations
DITTY: 28 valid combinations
DANDY: 28 valid combinations
```

A fourth letter option

Farmer John is considering adding a fourth letter to each of the five discs. What two extra words
should he choose to allow the letter lock to generate the maximum number of valid words?
NOTE: Number of flexible words trying here is 1000 to avoid crashing.

In [5]:
```
import cProfile
from itertools import product

# Load valid words into a set for fast membership checking
def load_valid_words(filename):
    with open(filename, 'r') as file:
        return {line.strip() for line in file}
```

```
# Count valid combinations using set intersection
def count_valid_combinations(fixed_row1, fixed_row2, flexible_row1, flexible_row2,
    # Generate all combinations for the given rows and store as a set
    generated_combinations = {
        ''.join(letters) for letters in product(fixed_row1, fixed_row2, flexible_ro
    }
    # Find the intersection with valid_words and count the matches
    valid_combinations = generated_combinations & valid_words
    return len(valid_combinations)

# Sequentially process word pairs to avoid parallel processing overhead
def find_best_two_words(valid_words, sample_size=50):
    fixed_row1, fixed_row2 = "BELLA", "LOVES"
    max_valid_combinations = 0
    best_combination = ()

    # Use a smaller subset for efficiency
    candidate_words = list(valid_words)[:sample_size]

    for i, word1 in enumerate(candidate_words):
        for word2 in candidate_words[i + 1:]:
            valid_count = count_valid_combinations(fixed_row1, fixed_row2, word1, w
            if valid_count > max_valid_combinations:
                max_valid_combinations = valid_count
                best_combination = (word1, word2)

    return best_combination, max_valid_combinations

# Main function with profiling to analyze performance
def main():
    valid_words = load_valid_words(r'C:\Users\s563104\OneDrive\Personal\valid_words
    best_words, max_count = find_best_two_words(valid_words, sample_size=1000)
    print(f"Best two words: {best_words} with {max_count} valid combinations")

if __name__ == "__main__":
    cProfile.run('main()')
```

```
Best two words: ('ROADS', 'MEANT') with 35 valid combinations
         1561946802 function calls (1561946609 primitive calls) in 623.053 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   499500  364.556    0.001  602.343    0.001 776448.py:10(count_valid_combinations)
        1    0.215    0.215    6.537    6.537 776448.py:20(find_best_two_words)
        1    0.000    0.000    6.538    6.538 776448.py:38(main)
        1    0.002    0.002    0.003    0.003 776448.py:5(load_valid_words)
        3    0.000    0.000    0.000    0.000 <frozen abc>:121(__subclasscheck__)
        1    0.000    0.000    0.000    0.000 <frozen codecs>:260(__init__)
        5    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:1390(_ha
ndle_fromlist)
        1    0.000    0.000    6.538    6.538 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 asyncio.py:200(_handle_events)
        1    0.000    0.000    0.000    0.000 asyncio.py:210(call_at)
        1    0.000    0.000    0.000    0.000 asyncio.py:225(add_callback)
        5    0.000    0.000    0.000    0.000 attrsettr.py:43(__getattr__)
        5    0.000    0.000    0.000    0.000 attrsettr.py:66(_get_attr_opt)
        1    0.000    0.000    0.000    0.000 base_events.py:1894(_add_callback)
       62    0.000    0.000    0.000    0.000 base_events.py:1904(_timer_handle_canc
elled)
      126    0.005    0.000 1172.811    9.308 base_events.py:1909(_run_once)
      250    0.000    0.000    0.000    0.000 base_events.py:2004(get_debug)
       62    0.000    0.000    0.000    0.000 base_events.py:447(create_future)
      126    0.000    0.000    0.000    0.000 base_events.py:539(_check_closed)
      253    0.000    0.000    0.001    0.000 base_events.py:734(time)
       63    0.000    0.000    0.001    0.000 base_events.py:743(call_later)
       63    0.000    0.000    0.001    0.000 base_events.py:767(call_at)
       63    0.000    0.000    0.001    0.000 base_events.py:785(call_soon)
       63    0.000    0.000    0.001    0.000 base_events.py:814(_call_soon)
        7    0.000    0.000    0.000    0.000 cp1252.py:22(decode)
        1    0.000    0.000    0.000    0.000 decorator.py:199(fix)
        1    0.000    0.000    0.063    0.063 decorator.py:229(fun)
       24    0.000    0.000    0.000    0.000 enum.py:1129(__new__)
       51    0.000    0.000    0.000    0.000 enum.py:1544(_get_value)
        9    0.000    0.000    0.000    0.000 enum.py:1551(__or__)
        8    0.000    0.000    0.000    0.000 enum.py:1562(__and__)
       24    0.000    0.000    0.000    0.000 enum.py:726(__call__)
       63    0.000    0.000    0.001    0.000 events.py:111(__init__)
        1    0.000    0.000    0.000    0.000 events.py:127(__lt__)
       62    0.000    0.000    0.001    0.000 events.py:155(cancel)
      126    0.001    0.000    0.001    0.000 events.py:36(__init__)
       62    0.000    0.000    0.000    0.000 events.py:72(cancel)
      126    0.001    0.000    0.010    0.000 events.py:86(_run)
       62    0.000    0.000    0.002    0.000 futures.py:315(_set_result_unless_canc
elled)
        1    0.000    0.000    0.063    0.063 history.py:55(only_when_enabled)
        1    0.004    0.004    0.060    0.060 history.py:833(_writeout_input_cache)
        1    0.000    0.000    0.000    0.000 history.py:839(_writeout_output_cache)
        1    0.000    0.000    0.063    0.063 history.py:845(writeout_cache)
        4    0.000    0.000    0.000    0.000 inspect.py:2792(name)
       10    0.000    0.000    0.000    0.000 inspect.py:2804(kind)
        1    0.000    0.000    0.000    0.000 inspect.py:2884(__init__)
        1    0.000    0.000    0.000    0.000 inspect.py:2892(args)
```

```
         1    0.000    0.000    0.000    0.000 inspect.py:2915(kwargs)
         1    0.000    0.000    0.000    0.000 inspect.py:2945(apply_defaults)
         4    0.000    0.000    0.000    0.000 inspect.py:3085(parameters)
         1    0.000    0.000    0.000    0.000 inspect.py:3129(_bind)
         1    0.000    0.000    0.000    0.000 inspect.py:3268(bind)
         1    0.000    0.000    0.000    0.000 interactiveshell.py:315(_modified_ope
n)
         2    0.000    0.000    0.000    0.000 ioloop.py:541(time)
         1    0.000    0.000    0.000    0.000 ioloop.py:596(call_later)
         1    0.000    0.000    0.000    0.000 ioloop.py:742(_run_callback)
        62    0.001    0.000    0.006    0.000 iostream.py:118(_run_event_pipe_gc)
        62    0.001    0.000    0.002    0.000 iostream.py:127(_event_pipe_gc)
         1    0.000    0.000    0.000    0.000 iostream.py:138(_event_pipe)
         2    0.000    0.000    0.000    0.000 iostream.py:157(_handle_event)
         1    0.000    0.000    0.000    0.000 iostream.py:213(_is_master_process)
         1    0.000    0.000    0.000    0.000 iostream.py:216(_check_mp_mode)
         1    0.000    0.000    0.000    0.000 iostream.py:255(closed)
         1    0.000    0.000    0.000    0.000 iostream.py:259(schedule)
         1    0.000    0.000    0.000    0.000 iostream.py:276(<lambda>)
         1    0.000    0.000    0.000    0.000 iostream.py:278(_really_send)
         2    0.000    0.000    0.000    0.000 iostream.py:505(parent_header)
         2    0.000    0.000    0.000    0.000 iostream.py:550(_is_master_process)
         2    0.000    0.000    0.000    0.000 iostream.py:577(_schedule_flush)
         1    0.000    0.000    0.000    0.000 iostream.py:587(_schedule_in_thread)
         2    0.000    0.000    0.000    0.000 iostream.py:655(write)
         3    0.000    0.000    0.000    0.000 queue.py:209(_qsize)
         3    0.000    0.000    0.000    0.000 queue.py:97(empty)
       126    0.000    0.000    0.000    0.000 selector_events.py:750(_process_event
s)
         1    0.000    0.000    0.000    0.000 selectors.py:275(_key_from_fd)
       126   19.397    0.154  616.433    4.892 selectors.py:313(_select)
       126    0.001    0.000  616.435    4.892 selectors.py:319(select)
         8    0.000    0.000    0.000    0.000 socket.py:626(send)
         1    0.000    0.000    0.000    0.000 socket.py:703(send_multipart)
         2    0.000    0.000    0.000    0.000 socket.py:774(recv_multipart)
       124    0.001    0.000    0.003    0.000 tasks.py:653(sleep)
       187    0.000    0.000    0.000    0.000 threading.py:1155(_wait_for_tstate_loc
k)
       187    0.001    0.000    0.001    0.000 threading.py:1222(is_alive)
         2    0.000    0.000    0.000    0.000 threading.py:299(__enter__)
         2    0.000    0.000    0.000    0.000 threading.py:302(__exit__)
         1    0.000    0.000    0.000    0.000 threading.py:308(_release_save)
         1    0.000    0.000    0.000    0.000 threading.py:311(_acquire_restore)
         1    0.000    0.000    0.000    0.000 threading.py:314(_is_owned)
       187    0.000    0.000    0.000    0.000 threading.py:601(is_set)
         1    0.000    0.000    0.000    0.000 threading.py:627(clear)
         1    0.000    0.000    0.000    0.000 traitlets.py:1512(_notify_trait)
         1    0.000    0.000    0.000    0.000 traitlets.py:1523(notify_change)
         1    0.000    0.000    0.000    0.000 traitlets.py:1527(_notify_observers)
         2    0.000    0.000    0.000    0.000 traitlets.py:2304(validate)
         2    0.000    0.000    0.000    0.000 traitlets.py:3474(validate)
         2    0.000    0.000    0.000    0.000 traitlets.py:3486(validate_elements)
         2    0.000    0.000    0.000    0.000 traitlets.py:3624(validate_elements)
         2    0.000    0.000    0.000    0.000 traitlets.py:3631(set)
         6    0.000    0.000    0.000    0.000 traitlets.py:629(get)
         6    0.000    0.000    0.000    0.000 traitlets.py:676(__get__)
```

```
        2    0.000    0.000    0.000    0.000 traitlets.py:689(set)
        2    0.000    0.000    0.000    0.000 traitlets.py:708(__set__)
        2    0.000    0.000    0.000    0.000 traitlets.py:718(_validate)
        2    0.000    0.000    0.000    0.000 traitlets.py:727(_cross_validate)
        3    0.000    0.000    0.000    0.000 typing.py:1221(__instancecheck__)
        4    0.000    0.000    0.000    0.000 typing.py:1285(__hash__)
        3    0.000    0.000    0.000    0.000 typing.py:1492(__subclasscheck__)
       24    0.000    0.000    0.000    0.000 typing.py:2182(cast)
        6    0.000    0.000    0.000    0.000 typing.py:392(inner)
        1    0.000    0.000    0.000    0.000 tz.py:74(utcoffset)
        5    0.000    0.000    0.000    0.000 zmqstream.py:538(receiving)
        3    0.000    0.000    0.000    0.000 zmqstream.py:542(sending)
        2    0.000    0.000    0.000    0.000 zmqstream.py:556(_run_callback)
        2    0.000    0.000    0.000    0.000 zmqstream.py:583(_handle_events)
        2    0.000    0.000    0.000    0.000 zmqstream.py:624(_handle_recv)
        3    0.000    0.000    0.000    0.000 zmqstream.py:663(_rebuild_io_state)
        3    0.000    0.000    0.000    0.000 zmqstream.py:686(_update_handler)
        1    0.000    0.000    0.000    0.000 zmqstream.py:694(<lambda>)
        3    0.000    0.000    0.000    0.000 {built-in method _abc._abc_subclassche
ck}
       63    0.000    0.000    0.000    0.000 {built-in method _asyncio.get_running_
loop}
        7    0.000    0.000    0.000    0.000 {built-in method _codecs.charmap_decod
e}
       64    0.000    0.000    0.000    0.000 {built-in method _contextvars.copy_con
text}
       62    0.000    0.000    0.000    0.000 {built-in method _heapq.heappop}
       63    0.000    0.000    0.000    0.000 {built-in method _heapq.heappush}
        1    0.000    0.000    0.000    0.000 {built-in method _io.open}
        1    0.000    0.000    0.000    0.000 {built-in method _thread.allocate_loc
k}
        1    0.000    0.000   36.659   36.659 {built-in method builtins.exec}
        5    0.000    0.000    0.000    0.000 {built-in method builtins.getattr}
       10    0.000    0.000    0.000    0.000 {built-in method builtins.hasattr}
        4    0.000    0.000    0.000    0.000 {built-in method builtins.hash}
  116/110    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.issubclass}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.iter}
   499762    0.141    0.000    0.141    0.000 {built-in method builtins.len}
      192    0.000    0.000    0.000    0.000 {built-in method builtins.max}
       64    0.000    0.000    0.000    0.000 {built-in method builtins.min}
        6    0.000    0.000    0.000    0.000 {built-in method builtins.next}
        1    0.000    0.000    0.001    0.001 {built-in method builtins.print}
        3    0.000    0.000    0.000    0.000 {built-in method nt.getpid}
      126    1.041    0.008   30.173    0.239 {built-in method select.select}
      253    0.000    0.000    0.000    0.000 {built-in method time.monotonic}
        2    0.000    0.000    0.000    0.000 {built-in method time.time}
        2    0.000    0.000    0.000    0.000 {method '__enter__' of '_thread.lock'
objects}
        1    0.000    0.000    0.000    0.000 {method '__exit__' of '_io._IOBase' ob
jects}
        2    0.000    0.000    0.000    0.000 {method '__exit__' of '_thread.RLock'
objects}
       69    0.000    0.000    0.000    0.000 {method '__exit__' of '_thread.lock' o
bjects}
        2    0.000    0.000    0.000    0.000 {method '__exit__' of 'sqlite3.Connect
```

```
ion' objects}
    190/3    0.000    0.000    0.000    0.000 {method 'acquire' of '_thread.lock' ob
jects}
      128    0.000    0.000    0.000    0.000 {method 'append' of 'collections.dequ
e' objects}
        5    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
       62    0.000    0.000    0.000    0.000 {method 'cancelled' of '_asyncio.Futur
e' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profile
r' objects}
        1    0.000    0.000    0.001    0.001 {method 'execute' of 'sqlite3.Connecti
on' objects}
        2    0.000    0.000    0.000    0.000 {method 'extend' of 'list' objects}
        2    0.000    0.000    0.000    0.000 {method 'get' of '_contextvars.Context
Var' objects}
        4    0.000    0.000    0.000    0.000 {method 'get' of 'dict' objects}
       64    0.000    0.000    0.000    0.000 {method 'items' of 'dict' objects}
        3    0.000    0.000    0.000    0.000 {method 'items' of 'mappingproxy' obje
cts}
1560937500  237.676    0.000  237.676    0.000 {method 'join' of 'str' objects}
      128    0.000    0.000    0.000    0.000 {method 'popleft' of 'collections.dequ
e' objects}
        1    0.000    0.000    0.000    0.000 {method 'release' of '_thread.lock' ob
jects}
      126    0.001    0.000    0.010    0.000 {method 'run' of '_contextvars.Contex
t' objects}
       62    0.000    0.000    0.002    0.000 {method 'set_result' of '_asyncio.Futu
re' objects}
     4981    0.001    0.000    0.001    0.000 {method 'strip' of 'str' objects}
        5    0.000    0.000    0.000    0.000 {method 'upper' of 'str' objects}
        1    0.000    0.000    0.000    0.000 {method 'values' of 'mappingproxy' obj
ects}
        2    0.000    0.000    0.000    0.000 {method 'write' of '_io.StringIO' obje
cts}
```

A fourth letter option

Variation

Let's try with a wordlist size of 50.

In [9]:
```python
# Load valid words from a file into a list
def load_valid_words(filename):
    valid_words = []
    with open(filename, 'r') as file:
        for line in file:
            valid_words.append(line.strip().upper())
    return valid_words

# Generate possible 5-letter combinations by rotating letters in each slot
def generate_combinations(fixed_row1, fixed_row2, flexible_row1, flexible_row2):
    all_combinations = []
```

```
    for i in range(5):
        for j in range(5):
            for k in range(5):
                for m in range(5):
                    for n in range(5):
                        word = fixed_row1[i] + fixed_row2[j] + flexible_row1[k] + f
                        all_combinations.append(word)
    return all_combinations

# Check if a combination is in the list of valid words
def count_valid_words(combinations, valid_words):
    count = 0
    for word in combinations:
        if word in valid_words:
            count += 1
    return count

# Find the best two flexible rows for maximum valid combinations
def find_best_two_words(valid_words, fixed_row1="BELLA", fixed_row2="LOVES", sample
    max_valid_combinations = 0
    best_words = (None, None)

    # Limit the candidates for flexible rows to the first 50 words
    candidate_words = valid_words[:sample_size]

    # Try each pair of words as flexible rows
    for i, word1 in enumerate(candidate_words):
        for j, word2 in enumerate(candidate_words):
            if word1 != word2:
                # Generate all combinations for the candidate word pair
                combinations = generate_combinations(fixed_row1, fixed_row2, word1,

                # Count how many of these combinations are in the valid words list
                valid_count = count_valid_words(combinations, valid_words)

                # Check if this pair has the most valid combinations
                if valid_count > max_valid_combinations:
                    max_valid_combinations = valid_count
                    best_words = (word1, word2)
                    print(f"New best combination found: {best_words} with {max_vali

    return best_words, max_valid_combinations

# Main function
def main():
    # Load valid words from file (assumes valid_words.txt in the same directory)
    valid_words = load_valid_words(r'C:\Users\s563104\OneDrive\Personal\valid_words

    # Find the best two words for the flexible rows
    best_words, max_valid_combinations = find_best_two_words(valid_words, sample_si

    print("Best words for the flexible rows:", best_words)
    print("Maximum valid combinations:", max_valid_combinations)

# Run the main function
```

```
if __name__ == "__main__":
    main()
```

```
New best combination found: ('Ï»¿WHICH', 'THERE') with 4 valid combinations
New best combination found: ('THERE', 'THEIR') with 7 valid combinations
New best combination found: ('THEIR', 'THERE') with 8 valid combinations
New best combination found: ('ABOUT', 'THERE') with 10 valid combinations
New best combination found: ('ABOUT', 'THESE') with 14 valid combinations
New best combination found: ('WORDS', 'GOING') with 15 valid combinations
New best combination found: ('RIGHT', 'AGAIN') with 19 valid combinations
New best combination found: ('GREAT', 'AGAIN') with 21 valid combinations
New best combination found: ('GOING', 'AGAIN') with 32 valid combinations
Best words for the flexible rows: ('GOING', 'AGAIN')
Maximum valid combinations: 32
```

Discussion

When designing the padlock solution using two fixed rows (BELLA and LOVES) and two flexible rows chosen from the list of valid words, we face a significant computational challenge. Here's a breakdown of the process, focusing on why generating and verifying these combinations is computationally demanding:
Combination Generation:

For each pair of flexible words, we generate combinations by rotating each of the five letters in each slot. Given that each letter slot in each row has four options (due to our flexible rows), we end up with

4×4×4×4×4=1024 unique 5-letter combinations per flexible word pair. Scale of Verification:

After generating these 1024 combinations, each combination needs to be checked against a valid_words list, which contains approximately 4980 words. This results in 1024×4980=5,098,240 individual comparisons for a single pair of flexible words.
Multiple Flexible Word Pairs:

Since we aim to test multiple pairs of flexible words to find the pair generating the maximum valid combinations, this process is repeated many times. For instance, if we choose a sample of 50 words as candidates for the flexible rows, we would end up testing approximately $50 \times 49/2 = 1225$ unique pairs. This leads to a total of 1225×5,098,240≈6,248,960,000 comparisons.
Resource and Time Requirements:

This calculation leads to a total of approximately 6.25 billion comparisons (1225 pairs × 5,098,240 combinations per pair). This staggering figure represents just 50 words.

If it is to scale this up to the full 4980-word list, we would be looking at over 600 billion comparisons. The computational demand highlights just how challenging it becomes to handle large datasets for combination generation and validation, making this approach daunting without significant optimization techniques.