

CSE 202

PROJECT REPORT

Wordle Bot

Team:

Sanjayan Sreekala
Rohith Sathiamoorthy Pandian
Samanvitha Sateesha
Aakriti Kedia
Karan Santhosh

March 18, 2023

Contents

1	Introduction	1
1.1	Background	1
1.1.1	Rules of the Game	1
1.2	Some unique cases to consider	3
2	Common Definitions	4
3	Problem 1: Finding V^n	5
3.1	Instance	5
3.2	Solution Format	5
3.3	Solution Constraint	5
3.4	Solution Objective	5
3.5	Algorithms	5
3.5.1	Brute Force approach	5
3.5.2	Hash table based approach	6
3.6	Implementation, Experiments and Results	9
3.6.1	Game play by Finding V^n - Intuition for Heuristic Algorithm	9
4	Problem 2: Strategies for finding the next best guess	11
4.1	Strategy 1: Calculating the word that reduces V^n for every possible win condition	11
4.1.1	Instance	11
4.1.2	Solution Format	11
4.1.3	Solution Constraint	11
4.1.4	Solution Objective	11
4.1.5	Brute Force Algorithm	11
4.1.6	Optimized Algorithm	13
4.1.7	Implementation, Experiments and Results	15
4.2	Strategy 2: Prioritizing Words based on Heuristics	17
4.2.1	Non-Duplicate Prioritization	17
4.2.2	Letter Frequency Same Index Prioritization	18
4.2.3	Letter Frequency Cross-Index Prioritization	20
4.2.4	Unattempted Letter Prioritization	21
4.2.5	Implementation, Experiments and Results	23
4.3	Comparison of Implementation of Strategies	24
5	Problem 3: Word Validity (Evaluation Bot)	26
5.1	Instance	26
5.2	Solution Format	26
5.3	Solution Constraints	26
5.4	Solution Objective	26
5.5	Algorithms	26
5.6	Trie based approach	26
5.6.1	Trie creation - Insert word	26
5.6.2	Search word	27
5.7	Brute Force	28

6	Problem 4: Evaluate guess word	30
6.1	Instance	30
6.2	Solution Format	30
6.3	Solution Constraints	30
6.4	Solution Objective	30
6.5	Algorithms	30
6.6	Time complexity analysis	31
7	Model Extensions	32

1 Introduction

In our project, we have decided to base our work on the popular web-based word game known as Wordle. The game’s primary objective is to guess a 5-letter word within a specific number of attempts. In each guess, we receive feedback in the form of colored indicators: green, yellow, and gray. The green color indicates that the letter we guessed is correct, and it is also in the right position. The yellow color indicates that the letter we guessed is correct but in the wrong position. Finally, the gray color indicates that the letter we guessed is incorrect and does not appear in the word. These feedback colors provide valuable information for players to narrow down their guesses and eventually guess the word correctly.

1.1 Background

The game of Wordle is a popular word guessing game that has recently been acquired by The New York Times Company. The game is played by guessing a five-letter word within six attempts, with feedback given for each guess in the form of colored tiles indicating when letters match or occupy the correct position. The game is similar to the 1955 pen-and-paper game Jotto and the game show franchise Lingo. The objective of the game is to guess the word correctly within the given attempts.

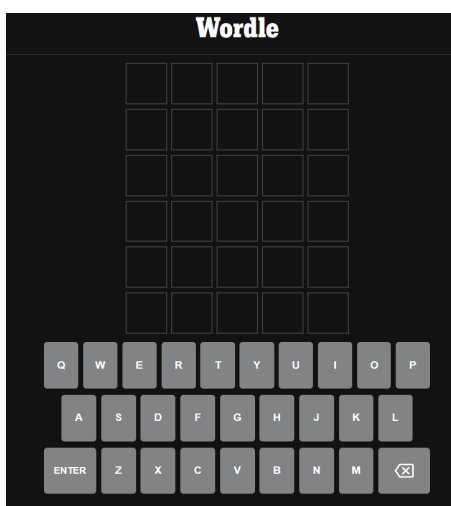


Figure 1: Caption

1.1.1 Rules of the Game

The game of Wordle is simple to understand and play. Players are presented with a set of letter tiles and must use them to guess a five-letter word within six attempts. Only valid English words are allowed to be guessed. After every guess, each letter is marked as either green, yellow or gray: green indicates that letter is correct and in the correct position, yellow means it is in the answer but not in the right position, while gray indicates it is not in the answer at all. The game also has a “hard mode” option, which requires players to include letters marked as green and yellow in subsequent guesses. The daily word is the same for everyone.

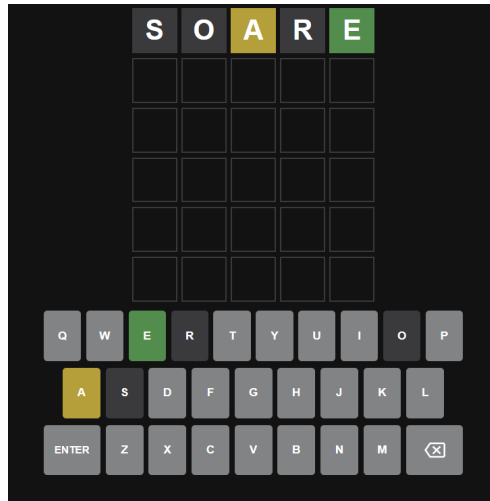


Figure 2: Caption

- The player's first attempt is "SOARE"
- The letter "A" is colored yellow, indicating it is in the word but not in the correct position.
- The letter "E" is colored green, indicating it is in the word and in the correct position.
- The rest of the letters "S", "O" and "R" are colored gray, indicating they are not in the word.
- The player now has 5 more attempts to guess the word and make use of the feedback given by the colored tiles to make better guesses.

Here is an example of the game where the player loses with the correct word being "MAIZE"

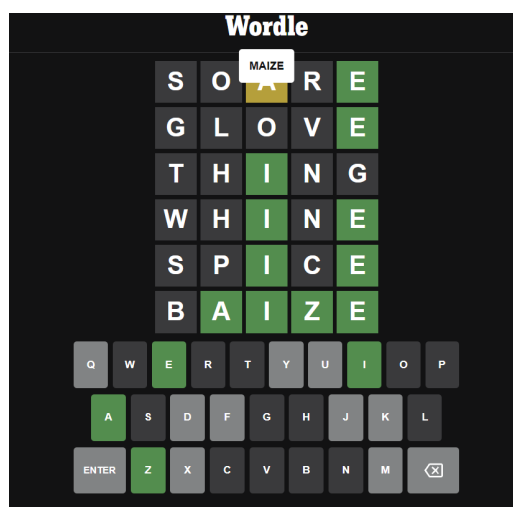


Figure 3: Caption

1.2 Some unique cases to consider

Consider, $n = 4$, $L = \{A, B, C, D, E, F, G\}$ and $W^4 = \{ABCD, ABCE, ABCF, ABCG, DEFG\}$. Assume we also know that the first 3 letters have to be ABC . Which implies $V^4 = \{ABCD, ABCE, ABCF, ABCG\}$. Intuitively we can see the next best guess would be $DEFG$ since that would narrow our guess after that to just one. Therefore we can win this game in 2 attempts. Notice that $DEFG \notin V^n$.

2 Common Definitions

Let the game state be represented by G , where G is the current board configuration. G can be represented as a $m \times n$ matrix, where m is the number of total acceptable attempts and n is the word length. Let $G = (g_{ij})$.

We will define the color c of each letter in the game as an integer.

- 0 represents black or not attempted
- 1 represents gray or a letter that is not in the solution word
- 2 represents yellow or a correct letter in the wrong position
- 3 represents green or a correct letter in the correct position

Let L be the valid character set.

$g_{ij} = (l, c)$ where $l \in L$ (valid character set) and c represents the color.

Let W^n be a set of all possible words of length n formed by some combination of letters from L .

Let $V^n \subseteq W^n$ be the set of possible win words. This subset is created with an intention of maximizing our chance of winning by guessing the correct word in the minimum number of attempts.

3 Problem 1: Finding V^n

After each turn in the game, we get new information about the final word. So the set of **possible win words** V^n will reduce in size. We need to compute this set of new words each time based on this information. The input available at each stage is the game state G and the current set of words. the only new information in the new game state is just the latest attempted word w_k and its evaluation result *colors* (i.e, the colors corresponding to each letter).

3.1 Instance

The algorithm receives a game state G and the set of all valid words of length n , W^n . $g_{ij} = (l, c)$.

3.2 Solution Format

A set V^n .

3.3 Solution Constraint

- $V^n \subseteq W^n$
- v doesn't contain any greyed letters
- v contains all yellow letters
- v contains all green letters in their correct positions

3.4 Solution Objective

To find the V^n , the possible win words, that contains only words from W^n based on new constraints imposed by the latest evaluation result.

3.5 Algorithms

Both the algorithms described below use a similar logic. When a letter is colored **GRAY** all words that have that letter can no longer be a win word and has to be removed. When a letter is colored **YELLOW** it means that words that have that letter at the given positions can no longer be a win word. Also the win word needs to have the letter at one of the other positions. Finally if the letter is colored **GREEN**, we only need to keep words that have the given letter at the given position.

3.5.1 Brute Force approach

The brute force approach to remove invalid words for this new state is outlined in the pseudo code below:

Algorithm 1: Find next V^n

Input : V^n (List of words), G
Output: updated V^n

```
1  $V^{tmp} \leftarrow V^n.copy()$ ;  
2  $word \leftarrow$  Get the latest word played from  $G$ ;  
3  $colors \leftarrow$  Get the latest evaluation result from  $G$ ;  
4 for  $current \in V^{tmp}$  do  
5   for  $i, letter \in enumerate(current)$  do  
6     if  $colors[i] == GRAY$  and  $word[i] \in current$  then  
7        $V^n.remove(current)$ ;  
8       break;  
9     else if  $colors[i] == YELLOW$  and  $(word[i] \notin current$  or  $letter ==$   
10       $word[i])$  then  
11        $V^n.remove(current)$ ;  
11       break;  
12     else if  $colors[i] == GREEN$  and  $letter \neq word[i]$  then  
13        $V^n.remove(current)$ ;  
14       break;  
15 return  $V^n$ ;
```

3.5.1.1 Time Complexity

The first loop runs for each of the word in the current valid vocabulary V^n and the second loop runs $n=5$ times, i.e, for the length of the word which is a constant for the Wordle game. This comes to a time complexity of $O(mn)$. Each remove operations within the loops are removing an item from list which is $O(m)$ where $m = |V^n|$ is the number of words in V^n . Thus, the total time complexity will be $O(m^2n)$ but since the number of letters n is a constant, we get $O(m^2)$.

3.5.2 Hash table based approach

To improve the time complexity of finding the updated set of possible win words, we use a hash table based approach. We do a pre-processing step where we organize the **full vocabulary** W^n in a Hash table H of size $(word\ length \times alphabet\ size)$ which is 5×26 for Wordle. Each cell H_{ij} in the table stores a set of words. H_{ij} corresponds to words that have the character j at the i^{th} position. This way when we need to remove words, we directly have access to the information needed. This pre-processing will be done only once on W^n and will be reused in all further V^n computations.

Algorithm 2: Pre-processing

Input : W^n
Output: The hash table H

- 1 $H \leftarrow A$ 5×26 array of empty sets;
- 2 **for** $word \in W^n$ **do**
- 3 **for** i in $range(5)$ **do**
- 4 $H[i][ord(word[i]) - ord('a')].add(word);$
- 5 **return** H ;

Once we have the hash H we find the the required subset V^n using the following algorithm:

Algorithm 3: Hash table based V^n computation

Input : V^n (List of words), G
Output: updated V^n

- 1 $word \leftarrow$ Get the latest word played from G ;
- 2 $colors \leftarrow$ Get the latest evaluation result from G ;
- 3 $n = 5$
- 4 **for** $i, letter \in enumerate(word)$ **do**
- 5 **if** $colors[i] == Colour.GRAY$ **then**
- 6 **for** j in n **do**
- 7 $toDelete \leftarrow H[j][ord(letter) - ord('a')];$
- 8 **for** $word \in toDelete$ **do**
- 9 $V^n.discard(word);$
- 10 **else if** $colors[i] == Colour.YELLOW$ **then**
- 11 $toDelete \leftarrow H[i][ord(letter) - ord('a')];$
- 12 $wordsContainingLetter \leftarrow [];$
- 13 **for** j in $range(n)$ **do**
- 14 **if** $j \neq i$ **then**
- 15 $wordsContainingLetter.extend(H[j][ord(letter) - ord('a')]);$
- 16 $wordsContainingCetter \leftarrow set(wordsContainingCetter);$
- 17 $V^n \leftarrow set.intersection(V^n, wordsContainingCetter);$
- 18 **for** $word \in toDelete$ **do**
- 19 $V^n.discard(word);$
- 20 **else**
- 21 $toDelete \leftarrow flatten(H[i][:ord(letter) - ord('a')]);$
- 22 $toDelete.extend(flatten(H[i][ord(letter) - ord('a')+1:]));$
- 23 **for** $word \in toDelete$ **do**
- 24 $V^n.discard(word);$
- 25 **return** V^n ;

3.5.2.1 Time Complexity

The time complexity of the Hash based algorithm depends on the length of the target word and the size of the vocabulary. Let n be the length of the target word, and let m be the size of the vocabulary. The algorithm iterates over each letter in the target word, so it has a time complexity of $O(n)$ for the outer loop.

For each letter, the algorithm either iterates over each word in a sub-list of the hash table or performs set intersection and set difference operations on the V^n set. The set of words that need processing can be obtained in $O(1)$ from the hash table. Each sub-list contains only a fraction of the total vocabulary, and the set operations are performed on a subset of the vocabulary, so the worst-case time complexity of each inner loop is $O(m)$. This is because the averaged amortized cost of all intersection set operation is $O(m)$ and discard/delete operations are $O(1)$.

Therefore, the overall time complexity is $O(nm)$. Since n is a constant for Wordle, we have a asymptotic time complexity bounded by $O(m)$. But in practice, the algorithm will likely run faster than this worst-case bound due to the performance of set operations and the fact that most words will be removed from V^n early in the algorithm. The actual time will be a small fraction of m . Experiments to show this will be included in the implementation report.

3.5.2.2 Correctness

The algorithm above is designed to update the set of valid words, V^n , after a player has played a new word. The goal is to remove from V^n all words that cannot be the target word based on the clues from the evaluation result. To prove the correctness of the algorithm, we need to show that:

- The algorithm correctly handles each of the three possible colors (gray, yellow, and red) and removes all words from V^n that are inconsistent with the evaluation result.
- $V^n \subseteq W^n$

To prove the first point, we can analyze each of the three color cases:

1. For the gray case, the algorithm loops over all words in H that contain the gray letter and removes them from V^n . This correctly handles the case where the gray letter appears in any position in the target word, since all words containing the gray letter are removed.
2. For the yellow case, the algorithm first creates a set of all words in H that contain the yellow letter in any position except the current position. It then takes the intersection of this set with V^n to find all words that are possible target words. Finally, it removes all words containing the yellow letter in the current position from V^n . This correctly handles the case where the yellow letter appears in any position except the current one, since it removes all words that don't contain the yellow letter in any position except the current one.
3. For the green case, we can observe that if a word does not contain the given letter in the specified position, it cannot be the target word. Therefore, removing such words from V^n is correct and does not remove any valid target words.

Finally since V^n was already a subset of W^n and we are only removing words from it, it is still a subset of W^n .

In conclusion, the algorithm above is correct and removes all words from V^n that are inconsistent with the clues provided by the evaluation result, while leaving only the words that are consistent with the clues and are still possible targets for the game.

3.6 Implementation, Experiments and Results

Both the the brute force approach and the Hash table based approach were implemented. Almost $2\times$ improvement was seen when using the Hash table based approach. The possible win word reduction was done on a vocabulary W^n of 14855 5-letter English words. The hash based method consistently performs better. The reduction was done by considering a random win word and a random guess word. This was done 1000 times and a running average was computed. This is shown in figure 4 below.

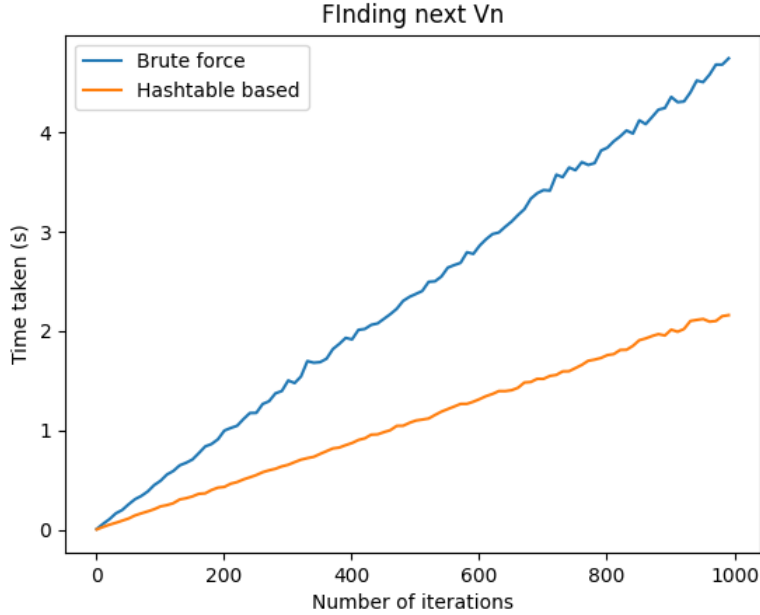


Figure 4: Finding V^n comparison

3.6.1 Game play by Finding V^n - Intuition for Heuristic Algorithm

When coming up with the algorithm to solve the Wordle game the complete solution is roughly bounded by $O(m^4)$ to get the absolute best solution, where m is the vocabulary size (Strategy 1, described in next section). The optimised version of the same still takes $O(m^3)$ which is quite large for a vocabulary size (14855 5-letter words). On analysing the game play using just the V^n reduction we see that frequency of occurrence of letters at given position plays a role in the size of V^n reduction. One example is shown in figures 5 and 6.



Figure 5: Game play

```
karansanthosh@Karans-MacBook-Pro wordle-bot % /usr/local/bin/python3 /Users/karansanthosh/Documents/CSE202/Project/wordle-bot/wordle_bot/player/Vn.py
Guess: slate
Vn:
{'speer', 'sperm', 'skein', 'skewy', 'spied', 'sebum', 'shied', 'sinew', 'speck', 'smeek', 'sorex', 'soken', 'smeer', 'semis', 'shend', 'speen', 'spe',
ed', 'sures', 'sward', 'sheik', 'sukey', 'sizer', 'swego', 'spyer', 'skeer', 'serow', 'skeep', 'shrew', 'sower', 'sider', 'shoer', 'shred', 'skies',
'shier', 'sexed', 'seepy', 'snerp', 'servo', 'sewer', 'serin', 'serio', 'sepic', 'sever', 'specs', 'seugh', 'skyey', 'serum', 'seder', 'skier', 'sere',
h', 'senso', 'seech', 'sewen', 'sheng', 'seric', 'shies', 'sweer', 'sewed', 'skeen', 'sepoy', 'seker', 'spend', 'skeeg', 'suevi', 'siper', 'seres',
'serif', 'sermo', 'spier', 'sides', 'sizes', 'sheer', 'sized', 'siver', 'serry', 'sekos', 'sprew', 'siren', 'soger', 'seism', 'sphex', 'sneck', 'sehyo',
'secos', 'senci', 'sedgy', 'sweep', 'suber', 'sixer', 'sober', 'sievy', 'spewy', 'speos', 'sheep', 'shyer', 'shemu', 'seven', 'sheen', 'snoek', 's',
eenu', 'skemp', 'serer', 'seedy', 'semen', 'scend', 'sided', 'skied', 'sedum', 'sneer', 'skeif', 'seron', 'super', 'screw', 'semic'}

Guess: shred
Vn:
{'sinew', 'smeek', 'soken', 'speen', 'sukey', 'skies', 'skyey', 'sewen', 'skeen', 'skeeg', 'sizes', 'sweep', 'seven', 'snoek', 'semen'}

Guess: sinew
Vn:
{'sweep'}
```

Figure 6: Finding V^n for each round

The reduction in V^n in game plays gives us intuition for the heuristic algorithm described as Strategy 2 detailed in next section.

4 Problem 2: Strategies for finding the next best guess

4.1 Strategy 1: Calculating the word that reduces V^n for every possible win condition

4.1.1 Instance

The algorithm receives the vocabulary words W^n and the set of possible win words V^n from Problem 1 in (3). $V^n \subset W^n$.

4.1.2 Solution Format

A single word b of length n from W^n

4.1.3 Solution Constraint

$b \in W^n$

4.1.4 Solution Objective

Let's assume we play a guess $g \in W^n$, then for every possible win word (say $w \in V^n$), we can update the game state. Now with this updated game state, we can calculate a new set of possible win words say V_2^n which would be smaller than V^n . Let $R(g, w) = |V^n| - |V_2^n|$ for a given combination of g and w . Now the objective is to find $\text{argmax}_{g \in W^n} (\sum_{w \in V^n} R(g, w))$.

4.1.5 Brute Force Algorithm

4.1.5.1 Algorithm Description - Brute Force

Using Wn to refer to W^n and Vn to refer to V^n

The algorithm starts by checking if there is only one possible winning word left in Vn . If so, it simply guesses that word. Otherwise, it initializes a list, $Vn_reduction_scores$, to keep track of the number of words that would be eliminated from Vn for each guess in Wn .

The algorithm then iterates through each possible guess in Wn and calculates the number of words in Vn that would be eliminated if that word were guessed for every possible win word. It does this by simulating the game state when guessing that word against each win word in Vn . We are running this algorithm assuming that each word in Vn can be a winning word. It keeps track of the number of words that would be eliminated for each guess and win word and updates the corresponding element in $Vn_reduction_scores$.

Finally, the algorithm guesses the word in Wn that corresponds to the maximum value in $Vn_reduction_scores$. This guess is expected to eliminate the largest number of words from Vn on average for any win word in Vn , thus increasing the chances of winning the game.

Algorithms for finding the next game state G when guessing $guess$ against win is described in 6.5.

Methods for reducing Vn are described in 3.

Algorithm 4: Calculating guess - Brute force

Input: Wn - set of all vocabulary words and Vn - set of all possible winning words
Output: Next guess

```

1 if  $|Vn| = 1$  then
2   | Guess the only word left in  $Vn$ ;
3 else
4   | Initialize  $Vn\_reduction\_scores$  as a list of length  $|Wn|$  filled with 0's;
5   | for  $i = 1$  to  $|Wn|$  do
6     | Let  $guess$  be the  $i^{th}$  word in  $Wn$ ;
7     | Initialize  $Vn\_reduction$  as 0;
8     | for each  $win$  in  $Vn$  do
9       | Let  $G$  be the game state when guessing  $guess$  against  $win$ ;
10      | Add the number of words in  $Vn$  that would be eliminated given  $G$  to
11      |    $Vn\_reduction$ ;
12      | Update the  $i^{th}$  element of  $Vn\_reduction\_scores$  to  $Vn\_reduction$ ;
13   | Guess the word in  $Wn$  corresponding to the maximum element of
14   |    $Vn\_reduction\_scores$ ;
15 End;

```

4.1.5.2 Correctness - Brute Force

The algorithm iterates through all possible guesses and win words and creates an updated game state for every such combination. It then uses each game state to see what the next V^n would be. Since the algorithms for finding the next game state G when guessing $guess$ against win reducing V^n is correct, this algorithm must correctly find the total reduction caused by each guess. This is $(\sum_{w \in V^n} R(g, w))$ described in the objective. Now the algorithm correctly finds the maximum by checking at each iteration if the guess is greater than the existing maximum. Hence the algorithm is correct.

4.1.5.3 Time Complexity - Brute Force

Let $m = |Vn|$. Let $w = |Wn|$.

For calculating a single guess we iterate through all possible guesses in Wn and through every win in Vn .

Calculating the game state is in the order of the word length n .

And for every game state, the reduction of Vn is in the order of $O(mn)$.

Now total time complexity is $O(wm * (n + mn))$ which is $O(wm^2n)$. Now we will assume n is a constant (like 5 in the original game). In that case, the time complexity is $O(wm^2)$.

Now we will look at the time complexity for all possible guesses for a game (note we will compare this against the optimal algorithm below). In the worst case, Vn can reduce by 1 after each guess (only the guessed word is eliminated).

Hence total time complexity $O(wm^2) + O(w(m-1)^2) + \dots + O(w.1^2)$. This is in $O(wm^3)$.

4.1.6 Optimized Algorithm

4.1.6.1 Algorithm Description

“reductions” and “reduction scores” refer to $R(g, w)$ and should be discernible from context.

Algorithm 5: Data structure Initialization

Input: Wn : set of all vocab words and Vn the set of all possible winning words

Output: Next Guess

```
1 Initialize guess_win_reductions removed2wins, and total_reductions as empty
  dictionaries;
2 for each guess  $\in Wn$  do
3   Initialize guess_win_reductions[guess] and removed2wins[guess] as empty
    dictionaries;
4   Initialize total_reductions[guess] as 0;
5   for each win  $\in Vn$  do
6     Compute the reduction r and the removed words removed_words in  $Vn$ 
      for the game state generated with guess and win;
7     Set guess_win_reductions[guess][win] to r;
8     Add r to total_reductions[guess];
9     for each removed_word in removed_words do
10      Add win to the set removed2wins[guess][removed_words];
11 best_guess =  $\operatorname{argmax}_{\text{guess}}(\text{total\_reductions}[\text{guess}])$ ;
12 return best_guess;
```

Algorithm 6: Data structure Update

Input: Wn , Data structures from Algorithm 5 and last row of latest game state

Output: Next guess

```
1 deleted_words = set of words inconsistent with the current guess row;
2 for each guess  $\in Wn$  do
3   for each deleted word in deleted_words do
4     Subtract guess_win_reductions[guess][deleted] from
      total_reductions[guess];
5     Remove guess_win_reductions[guess][deleted];
6     for each win in removed2wins[guess][deleted] do
7       if win  $\in$  deleted_words then
8         Remove win from removed2wins[guess][deleted];
9       else
10        Subtract 1 from guess_win_reductions[guess][win];
11        Subtract 1 from total_reductions[guess];
12 best_guess =  $\operatorname{argmax}_{\text{guess}}(\text{total\_reductions}[\text{guess}])$ ;
13 return best_guess;
```

This algorithm consists of 2 parts. Initializing some data structures for the first guess and then finding the guess for every subsequent board update. The idea is that for every subsequent guess, we might be able to reduce some run time by using appropriate data structures and updating them cleverly.

“Data structure Initialization” 5 is similar to the brute force solution. The difference is we keep track of all reductions for each *guess* and *win* and also the total reduction for each *guess*.

For every *guess* we also keep track of which *win* is causing the word to be removed.

Once we have this data structure we can check which *guess* has the most reduction to find the best *guess*.

Now for every “Data Structure Update” we are updating the data structure with a game state (for subsequent guesses) so that all the data structures will correctly maintain the values for the new Vn .

This is done by first reducing the data structure keeping track of total reductions of each *guess* by the reductions of each *guess*, *win* combination.

Further, every removed word from Vn might be already counted in the existing words “reduction” counts. Ie some word that is in Vn which is not deleted might be counting a deleted word in its reduction counts. We update the corresponding *guess* reduction count and *guess*, *win* reduction count.

Once we have the updated data structure we can check which *guess* has the most reduction to find the best *guess*.

4.1.6.2 Correctness - Optimal Solution

“Data structure Initialization” 5 is similar to the brute force solution. And is correct as 4.1.5.2 is correct. Once we have the data structures containing the reduction score for each guess, it is evident that we can trivially find the correct guess (arg max of this data structure).

Now we need to only show the data structure update is correct for the new Vn after the game state update since the *guess* with the maximum reduction is calculated from this.

Let the guess reduction count $R'(g) = \sum_{w \in Vn} R(g, w)$ can only change in 2 ways for a game state change. One, for some *guess* some corresponding *win* could be deleted. ie $R'(g) \leftarrow R'(g) - \sum_{w \in deleted} R(g, w)$. Two, for some *guess* and each *win* that is not deleted, the corresponding reduction score might decrease because the words that were getting counted in the reduction score for the *guess-win* ($R(g, w)$) might be deleted in this game state. Ie, this is a decrease in the value of $R(g, w)$ itself for some g and w . Now there is no other way for $R'(g)$ to reduce since $R'(g) = \sum_{w \in Vn} R(g, w)$.

Now since in the algorithm we are correctly reducing these values for every game state update, the new $R'(g)$ should be correct for the new Vn after every game state change.

Hence this algorithm is correct.

4.1.6.3 Time Complexity - Optimal Solution

Let $m = |Vn|$. Let $w = |Wn|$.

Data structure initialization is similar to the brute force solution 4.1.5.2. Additionally, we are storing the reduction values in each iteration of the loop. This can be done in $O(1)$ if we use hash-maps and hash-set data structures. We iterate over all the removed words and update the corresponding hash map for each *guess* and *win*. This is at most $O(m)$ (removed words from Vn can be at most $O(|Vn|)$) for each *guess* and *win*.

So time complexity for this is $O(wm(m + m)) \sim O(wm^2)$. Same as brute force.

Finding the best guess from the created data structures is an additional $O(w)$.

Now consider the case for subsequent guesses. Let $r = |\text{removed_words}|$, where *removed_words* is the set of words that are getting removed for each attempt on the game.

The update algorithm 6 iterates over each *guess* and each deleted word. For each deleted word it checks all the *win* words that could have caused this word to be deleted. The number of words that could have caused the deletion is at most $|Vn| = m$. Since the rest of the operations with hash-maps and hash-sets are in $O(1)$ the total time complexity is in $O(wrm)$.

Finding the best guess from the created data structures is an additional $O(w)$.

Now consider the subsequent guesses.

Again here m can reduce by at worst 1 for each guess similar to 4.1.5.2 Let r_1, r_2, \dots denote the number of words removed from Vn at each attempt (guess).

Now we have total time complexity at most $O(wm^2) + O(wr_1m) + O(wr_2(m-1)) + \dots O(wr_k1)$

This is bounded by $O(wm^2) + O(wr_1m) + O(wr_2m) + \dots + O(wr_km)$. This is equivalent to $O(wm^2) + O(wm(r_1 + r_2 + r_k))$.

Now note that the total sum of removals from Vn , $r_1 + r_2 + r_k$ is bounded by $|Vn|$. i.e. $r_1 + r_2 + r_k = m$. This is because you can only remove a total of $m(|Vn|)$ elements from Vn .

The time complexity then is $O(wm^2) + O(wm^2)$ which is $O(wm^2)$.

So the optimal solution has a better time complexity if we consider all the guesses that it will make for a game.

4.1.7 Implementation, Experiments and Results

Since we can only see a difference in run time of these 2 algorithms in the worst case scenario of one word being removed at each attempt, we simulate games with one being removed at a time to compare the algorithms.

Since the algorithms are in $O(wm^3)$ and $O(wm^2)$ we had to run for smaller w and m .

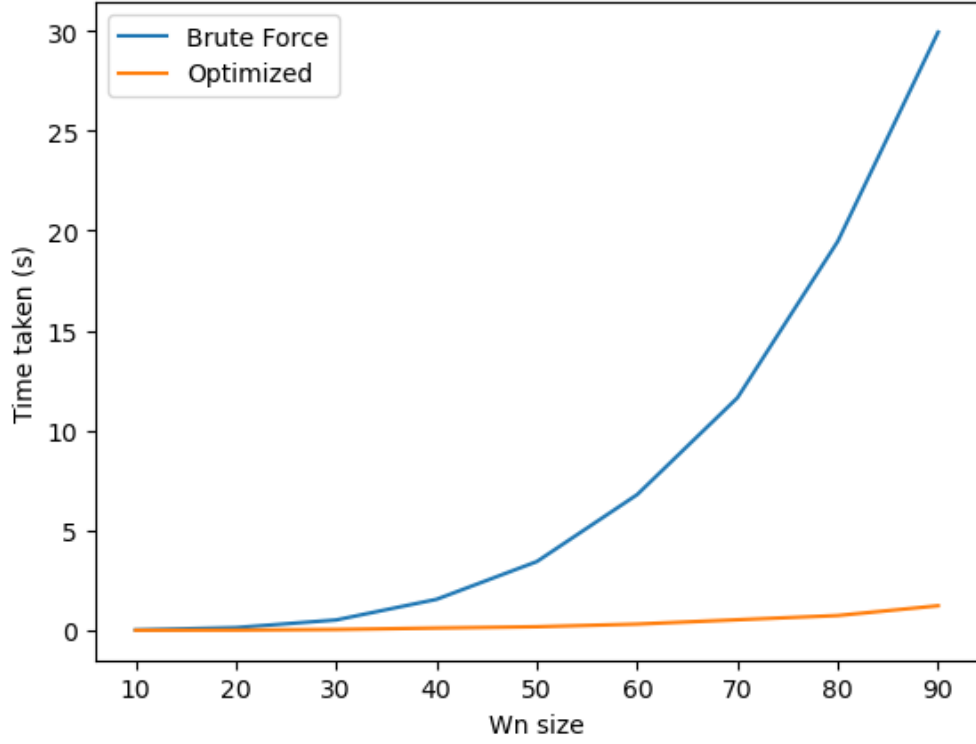


Figure 7: Time Taken vs $|Wn|$ for brute force and Optimized algorithms

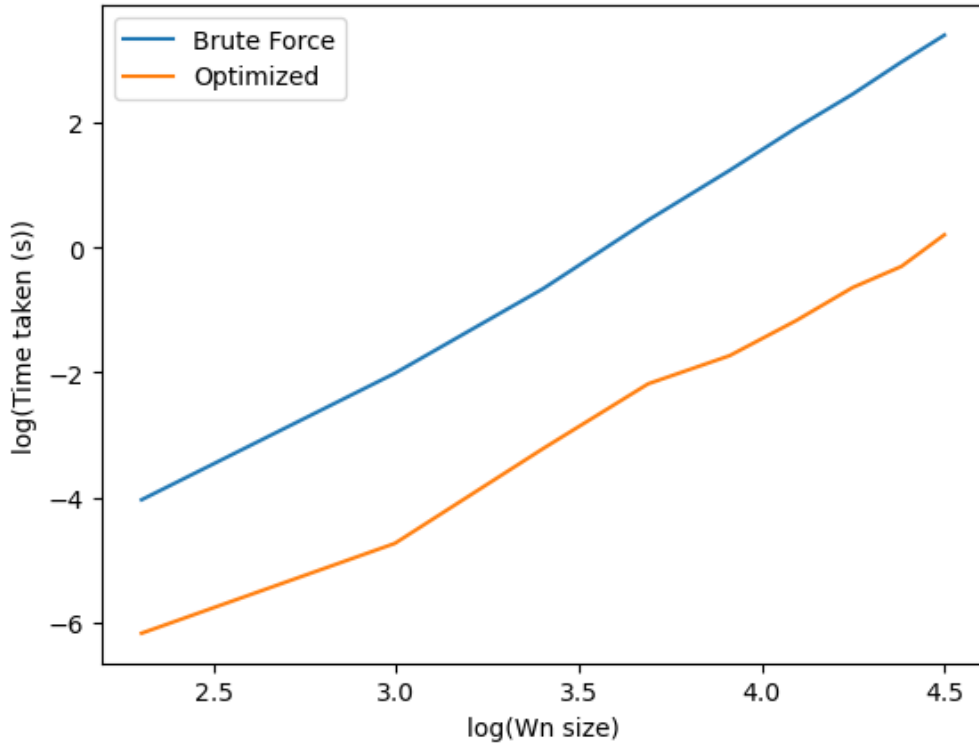


Figure 8: Time Taken vs $|Wn|$ for brute force and Optimized algorithms in log-log

Calculated slopes:

Slope Brute Force Algorithm: 3.434

Slope Optimized Algorithm: 2.953

We see that this is as expected. $O(wm^2)$ is bounded by $O(w^3)$ so slope of optimized algorithm is ≈ 3 .

The brute for algorithm is bounded by the $O(wm^3) \approx (w^4)$. If we have more samples it might approach 4. (however the time complexity is slightly lesser than 4 and only upper bounded by it).

4.2 Strategy 2: Prioritizing Words based on Heuristics

As part of our Wordle bot project, we have developed four prioritization algorithms to help our bot make educated guesses for the 5-letter word. The Non-Duplicate Prioritization algorithm (4.2.1) sorts the words based on the number of duplicates of each word in the given indexes, while the Letter Frequency Same Index Prioritization algorithm (4.2.2) prioritizes the words based on the frequency of letters at given indexes in each word. The Letter Frequency Cross-Index Prioritization algorithm (4.2.3) sorts the words based on the frequency of letters across all indexes of each word. The Unattempted Letter Prioritization (4.2.4) gives the word that contains the maximum number of unattempted letters to maximize our chances of winning and eliminate more letters in any given step.

These algorithms provide our Wordle bot with the ability to make informed guesses, increasing its chances of successfully guessing the 5-letter word within the allotted attempts.

4.2.1 Non-Duplicate Prioritization

4.2.1.1 Instance

The algorithm receives the list of valid words V^n after each guess and provides the best possible win word $word \subset V^n$.

4.2.1.2 Solution Format

A single word b of length n from V^n

4.2.1.3 Solution Constraint

$$b \subset V^n$$

4.2.1.4 Solution Objective

We need to find the best guess g from V^n and if it is not then it would calculate the new set of possible words V_2^n which would be smaller than V^n . Let $R(w) = \sum_{i,j \in \text{indexes} \setminus i \neq j} [w_i = w_j]$, be a function that returns the number of duplicate letters in the word w . The objective is to find $\text{argmin}_{w \in V^n} R(w)$, i.e., the word that has the minimum number of duplicates with all other words in the list. If there are ties then we will use Letter Frequency Same Index Prioritization Algorithm (4.2.2) with the tied words.

4.2.1.5 Algorithm

The Non-Duplicate Prioritization algorithm (7) prioritizes words in a list based on the number of duplicates of each word in the given indexes. Here, indexes denotes the

letter positions. It counts the number of unique characters in each word at the specified indexes and stores the counts in a list. Then, it sorts the words based on the values in the count list, so words with fewer duplicates are sorted first and returns the first word in the list.

Algorithm 7: Non-Duplicate Prioritization

Input: List of valid words V^n , List of indexes *indexes*

Output: List of words sorted based on the number of duplicates in the given indexes of each word

```

1 Initialize an empty list num_duplicates
2 for each word in  $V^n$  do
3   Initialize an empty set duplicates
4   for each index i in indexes do
5     letter = word[i]
6     if the letter in duplicates then
7       continue
8     Add the letter to duplicates;
9   num_duplicates.append(len(duplicates))
10 Sort  $V^n$  based on the values in num_duplicates in ascending order
11 return  $V^n[0]$ 

```

The algorithm helps in the game by prioritizing words that have fewer duplicates of the same letter in the given indexes. In other words, the algorithm sorts the words based on the number of unique letters used in the given indexes. This approach can be beneficial in reducing the size of the word search space (V^n) and helping the bot to guess the word more efficiently in subsequent rounds.

4.2.1.6 Time complexity analysis

The time complexity of this algorithm is $O(mn)$, where n is the number of the valid words V^n , i.e. $m = |V^n|$ and m is the length of the word (indexes list). The nested loops iterate through each word and each index (letter), and each iteration checks whether a character is already in the duplicates set. The set lookup takes $O(1)$ time on average, so the overall time complexity of the algorithm is $O(nm)$.

4.2.2 Letter Frequency Same Index Prioritization

4.2.2.1 Instance

The algorithm receives the list of valid words V^n after each guess and provides the best possible win word $word \in V^n$.

4.2.2.2 Solution Format

A single word b of length n from V^n

4.2.2.3 Solution Constraint

$b \in V^n$

4.2.2.4 Solution Objective

We need to find the best guess g from V^n and if it is not then it would calculate the new set of possible words V_2^n which would be smaller than V^n . Let $R(w) = \frac{1}{|indexes|} \sum_{i \in indexes} f_{i,w_i}$ where $f(i, w_i) = \sum_{w \in V^n} w_i$ and w_i is the i^{th} letter of word the word w . The function $f(i, w_i)$ stores the frequency of letter w_i at index i . The objective is to find $\text{argmax}_{w \in V^n} R(w)$. If there are ties then we will use Letter Frequency Cross Index Prioritization Algorithm (4.2.3) with the tied words.

4.2.2.5 Algorithm

The Letter Frequency Same Index Prioritization algorithm (9) is another algorithm useful for the wordle game that sorts a list of valid words based on the frequency of letters at the same index for each word. The algorithm initializes an empty dictionary *freq_dict* to store the frequency of each letter at the given indexes for each word. It then iterates over each word and each index in the list of indexes, incrementing the count of the letter in the frequency dictionary. Next, it calculates the average frequency of the letters for each word by summing up the frequency of each letter in the word at the given indexes and dividing by the number of indexes. Finally, it sorts the list of words based on their average frequency in descending order and returns the first word in the list.

Algorithm 8: Letter Frequency Same Index Prioritization

Input: List of valid words V^n , List of indexes *indexes*

Output: List of words sorted by average frequency of letters at given indexes

- 1 Initialize an empty dictionary *freq_dict* to store the frequency of each letter in the indexes for each word;
 - 2 **for** each word in V^n **do**
 - 3 **for** each i in *indexes* **do**
 - 4 $letter = word[i]$;
 - 5 $freq_dict[i][letter] = freq_dict[i].get(letter, 0) + 1$;
 - 6 Initialize an empty list *avg_freq* to store the average frequency of letters for each word;
 - 7 **for** each word in V^n **do**
 - 8 $total_freq = 0$;
 - 9 **for** each i in *indexes* **do**
 - 10 $total_freq = total_freq + freq_dict[i].get(word[i], 0)$;
 - 11 $avg_freq.append(total_freq/len(indexes))$;
 - 12 Sort the words based on their average frequency in descending order
 - 13 **return** $V^n[0]$
-

Like the Non-Duplicate Prioritization algorithm, this prioritization algorithm also helps reduce the size of the words by sorting them based on their average frequency of letters at given indexes. By sorting the list of valid words based on their average frequency in descending order, this algorithm provides the player with a prioritized list of words V^n with words in top that are more likely to match the hidden word in the game, reducing the number of guesses the bot has to make in subsequent rounds. This is a greedy strategy which will help reduce the vocab size and then we will apply more sophisticated algorithms to guess the correct word in less than 5 number of guesses.

4.2.2.6 Time complexity analysis

The time complexity of this algorithm is $O(mn + m \log m)$, where m is the number of valid words in V^n , i.e. $m = |V^n|$ and n is the number of indexes. The nested loops used to count the frequency of letters at given indexes and to calculate the average frequency of letters for each word have a time complexity of $O(mn)$. Sorting the words using Python's built-in sorted function has a time complexity of $O(m \log(m))$.

4.2.3 Letter Frequency Cross-Index Prioritization

4.2.3.1 Instance

The algorithm receives the list of valid words V^n after each guess and provides the best possible win word $word \subset V^n$.

4.2.3.2 Solution Format

A single word b of length n from V^n

4.2.3.3 Solution Constraint

$$b \subset V^n$$

4.2.3.4 Solution Objective

We need to find the best guess g from V^n and if it is not then it would calculate the new set of possible words V_2^n which would be smaller than V^n . Let $R(w) = \frac{1}{|indexes|} \sum_{i \in indexes} f_{w_i}$ where $f(w_i) = \sum_{w \in V^n} \sum_{i \in indexes} w_i$ and w_i is the i^{th} letter of word the word w . The function $f(i, w_i)$ stores the frequency of letter w_i at index i . The objective is to find $\argmax_{w \in V^n} R(w)$. If there are ties then we will use Unattempted Prioritization Algorithm (4.2.4) with the tied words.

4.2.3.5 Algorithm

The Letter Frequency Cross-Index Prioritization algorithm (9) is a prioritization algorithm that can sort a list of words based on the frequency of letters across all indexes of each word. The algorithm is designed to work by initializing an empty dictionary called *freq_dict* to store the frequency of each letter across all indexes for each word. The algorithm then iterates over each word and each index in the list of indexes, incrementing the count of the letter in the frequency dictionary. Next, it calculates the average frequency of the letters for each word by summing up the frequency of each letter in the word for the given indexes and dividing by the number of indexes. Finally, it sorts the list of words based on their average frequency in descending order and returns the first word in the list.

Algorithm 9: Letter Frequency Cross-Index Prioritization

Input: List of valid words V^n , list of indexes $indexes$

Output: List of words sorted by frequency of letters across all indexes

```
1 Initialize an empty dictionary freq_dict
2 for each word in  $V^n$  do
3     for each index  $i$  in  $indexes$  do
4         letter = word[i]
5         Increment the count of letter in freq_dict by 1
6 Initialize an empty list avg_freq
7 for each word in  $V^n$  do
8     total_freq = 0;
9     for each  $i$  in  $indexes$  do
10        total_freq = total_freq + freq_dict[i].get(word[i], 0);
11    avg_freq.append(total_freq/len( $indexes$ ));
12 Sort the words based on their average frequency in descending order
13 return  $V^n[0]$ 
```

This prioritization algorithm is useful for the wordle game as it can reduce the number of guesses the bot has to make by sorting the list of valid words based on their average frequency of letters at given indexes. By determining the frequency of letters at given indexes for each word in a list of valid words, the algorithm can calculate the average frequency of those letters across the indexes for each word. By sorting the list of valid words based on their average frequency in descending order, the algorithm can provide the bot with a smaller list of words V^n with words in top that are more likely to match the hidden word in the game and if not a match would significantly help reduce the size of the valid words V^n set for subsequent words.

4.2.3.6 Time complexity analysis

The time complexity of this algorithm is $O(mn + m \log m)$, where m is the number of valid words V^n , i.e. $m = |V^n|$ and n is the number of indexes in the list. This is because the algorithm iterates over each word and each index in the list, and performs a constant amount of work for each iteration which has a time complexity of $O(mn)$. The sorting step has a time complexity of $O(m \log m)$ using a typical sorting algorithm.

4.2.4 Unattempted Letter Prioritization

4.2.4.1 Instance

The algorithm receives the list of valid words V^n after the current guess, the list of all *valid_letters*, from all the guesses until this stage, and a list of all possible words, W^n and provides the best possible word $best_word \subset W^n$.

4.2.4.2 Solution Format

A single word b of length n from W^n

4.2.4.3 Solution Constraint

$$b \subset W^n$$

4.2.4.4 Solution Objective

We need to find the best guess g from W^n . Let v be the input *valid_letters*. Let $R(w) = \forall l \in w k(l)$ where $k(l)$ is a set of all $l \notin v$ where w is a word in V^n . Let $G(w^1) = \text{argmax} \sum_{l \in w^1} \text{count}(l \in R(w))$ where w^1 is a word in W^n . w^1 is the word that contains the highest number of unattempted letters. $b = w^1$. If there are ties, we will break them arbitrarily.

4.2.4.5 Algorithm

Algorithm 10: Unattempted Letter Prioritization

Input: List of vocab W^n , List of valid words V^n , List of letters that are either green or yellow *valid_letters*

Output: A word containing letters that have not been attempted yet

```

1 Initialize an empty list res_words;
2 Initialize an empty set possible_letters;
3 for each word in  $V^n$  do
4   for each letter in word do
5     if the letter not in valid_letters then
6       possible_letters.append(letter)
7 for each word in  $W^n$  do
8   count = 0
9   for each letter in word do
10    if letter in possible_letters then
11      count += 1
12  res_words.append((count, word));
13 Sort res_words based on the first index of res_words in descending order.
14 return First word of res_words

```

The Unattempted Letter Prioritization algorithm prioritizes words in a list based on the number of letters that have not been attempted in a particular word w , that belongs to vocab, W^n .

Let us assume a scenario where a word is of length 5 and we have guessed 3 out of 5 letters correctly. That is we have got green colour for 3 letters. We have to guess the remaining 2 letters. The updated V^n after getting 3 green letters contains around 10 words with equal probability. In such a scenario, if we have around 13 unattempted letters, we can increase our chances of winning by attempting a word that consists of the highest number of letters that haven't been attempted yet rather than a word consisting of all the green letters. We are not taking gray letters into account because V^n would have already eliminated all those words containing gray letters and those words will therefore not be included in *possible_letters*.

To achieve this, we get the input *valid_letters* that contains all the letters that are either yellow or green. We initialize two empty lists *res_words* and *possible_letters*. List

res_words stores the output words and list *possible_letters* consists of all the possible unattempted letters.

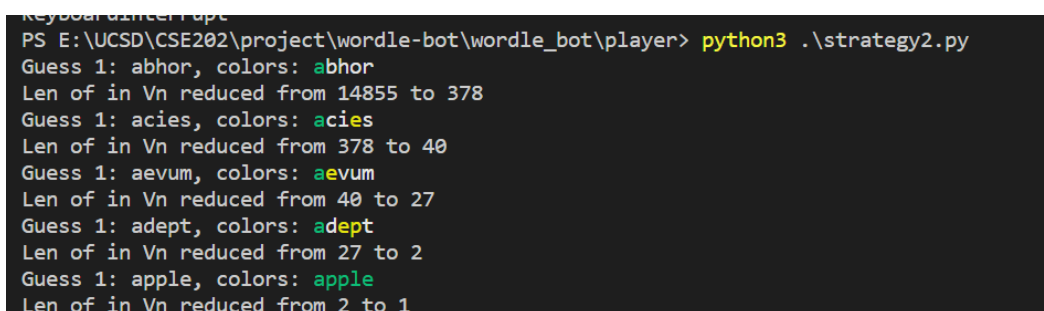
V^n consists of all the words that we have received as input after the recent update to the board state, i.e., the reduced vocab after getting all the *valid_letters*. We iterate through each word in V^n and add all those letters that are not in *valid_letters* to our list *possible_letters*. Once we have got the list *possible_letters*, we iterate through every word in W^n , and count the number of *possible_letters* in that word and append a tuple consisting of the count, and the word to *res_words*. We sort *res_words* based on the count in descending order and return the first word. If we have ties, we break it arbitrarily.

4.2.4.6 Time complexity analysis

The time complexity of this algorithm is $O(mn + kn + i \log i)$. It takes $O(mn)$ time for running the first loop - m denotes the length of V^n and n denotes the length of the word. It takes $O(kn)$ time for running the second loop of the algorithm - k denotes the length of W^n and n denotes the length of the word. It takes $i \log i$ time to sort *res_words* where i denotes the length of *res_words*. Here, as V^n is the subset of valid words and W^n is the actual vocabulary without any word reductions. The length of word, n will be a constant for Wordle (like 5 in the original game). In that case, the time complexity will be $O(m + k + i \log i)$.

4.2.5 Implementation, Experiments and Results

After implementing the heuristics and applying strategy 2 to play the game, we generated a histogram plot as shown in Figure 10 to visualize the relationship between the number of guesses and the number of words that took that corresponding number of guess. The plot shows that on average, it takes around 5 guesses to guess the word correctly, which is consistent with the maximum number of steps allowed in the online version of Wordle. In Figure 9, the Player Bot is shown guessing the word “apple” using Strategy 2. The figure provides a visual representation of the guessing process, demonstrating the effectiveness of the strategy in finding the correct word.



```

KeyboardInterrupt
PS E:\UCSD\CSE202\project\wordle-bot\wordle_bot\player> python3 .\strategy2.py
Guess 1: abhor, colors: abhor
Len of in Vn reduced from 14855 to 378
Guess 1: acies, colors: acies
Len of in Vn reduced from 378 to 40
Guess 1: aevum, colors: aevum
Len of in Vn reduced from 40 to 27
Guess 1: adept, colors: adept
Len of in Vn reduced from 27 to 2
Guess 1: apple, colors: apple
Len of in Vn reduced from 2 to 1

```

Figure 9: Player bot guessing the word “apple” using Strategy 2

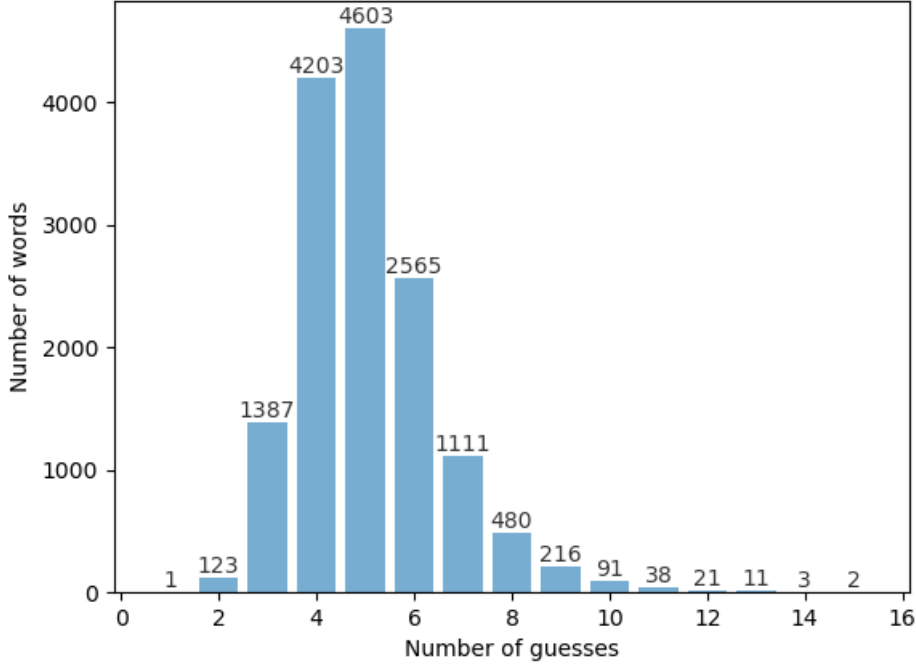


Figure 10: Number of words histogram for number of guesses

4.3 Comparison of Implementation of Strategies

We sampled 200 words randomly from Wn and randomly chose a correct word from these 200 words and computed the number of guesses it takes both Strategy 1 and Strategy 2 to win the game. This was done 200 times. The results obtained from this experiment were plotted as shown in 11 below. We also saw that the average number of guesses for strategy 1 was 2.73 and 2.78 for strategy 2. Since the size of Wn is less, the difference between the values of average guesses for strategy1 and strategy2 is minimal but still, strategy 1 is slightly better. We would see a significant difference of bigger Wn . However, the runtime of strategy1 is much higher (quadratic in the size of Wn). That is why we proposed strategy 2 which is a good alternative with a faster runtime (linear).

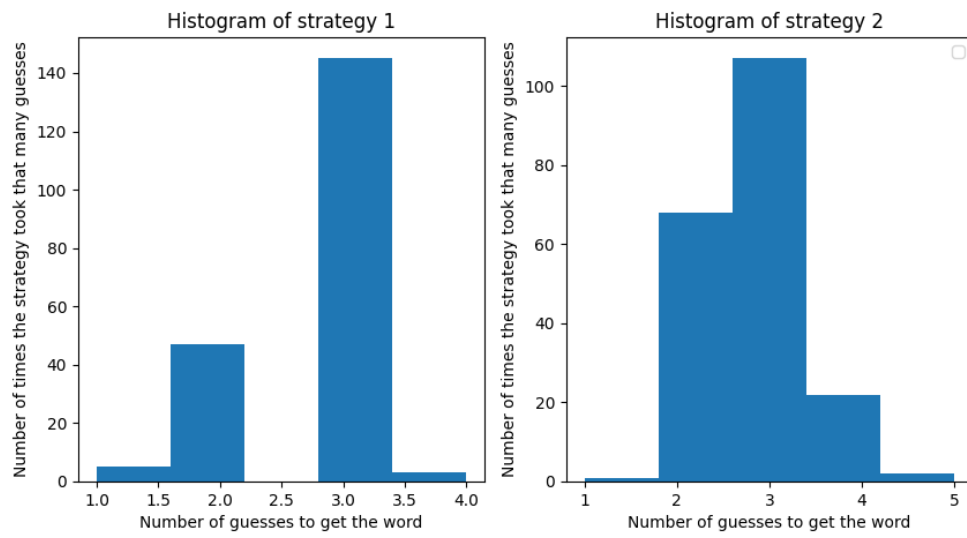


Figure 11: Comparison of Strategy 1 v/s Strategy 2

5 Problem 3: Word Validity (Evaluation Bot)

The evaluation bot checks if the word chosen by the player is a valid word, and accepts or rejects the input based on validity.

5.1 Instance

The algorithm receives a word w_i that the user (or player bot) inputs at the current game state and the set of all valid words W^n of length n each (Predefined vocabulary). The word w_i has to be of fixed length n .

5.2 Solution Format

A Boolean value '*isValid*' to indicate validity of w_i .

5.3 Solution Constraints

- Length of $w_i = n$
- *True* if $w_i \in W^n$ *False* otherwise.

5.4 Solution Objective

To check validity of w_i by comparing it against all words in the vocabulary W^n .

5.5 Algorithms

For this sub-task, we are using a trie data structure. The trie stores the set of all valid vocabulary and searches for the guessed word. We compare this against the brute force solution which will match the guessed word against each possible word in the corpus.

5.6 Trie based approach

5.6.1 Trie creation - Insert word

The Trie data structure is implemented using a tree-based approach, where each node represents a letter in a word. Each node in the Trie has an array of child nodes, where each index in the array represents a possible next letter in the word.

The insert word algorithm takes a word as input and adds it to the Trie data structure.

Algorithm 11: Insert Word using Trie

Input: Trie data structure *self*, word to insert *word*

Output: None

```
1 node ← self.root
2 for letter in word do
3   if letter not in node.children then
4     node.children[letter] ← new Node();
5   node ← node.children[letter];
6 node.term ← word;
```

5.6.1.1 Proof of Correctness:

As trie is a well-known data structure, we will use it and mark all vocab words as valid words by marking the *term* of last letter of each word as True.

5.6.1.2 Time complexity analysis

Let n be the length of the longest word in the vocabulary and m be the total number of words in the vocabulary.

When inserting a word into the Trie, we need to traverse the Trie from the root to the leaf that corresponds to the last letter of the word. This takes $O(n)$ time in the worst case. At each node, we need to check if a child node exists for the current letter, and if not, create a new node. This takes constant time, $O(1)$.

Since we need to insert each word in the vocabulary into the Trie, the total time complexity of building the Trie is given by the sum of the time complexity of inserting each word, which is $O(n)$ per word. Therefore, the overall time complexity of building the Trie is $O(mn)$.

In practice, the actual time complexity of building the Trie may be lower than $O(mn)$ if there are many common prefixes among the words, since these common prefixes can be shared among multiple words in the Trie. However, in the worst case where there are no common prefixes, the time complexity of building the Trie is still $O(mn)$. But we will be building the trie only once, and hence the game evaluation will be dominated by the time complexity of search word.

5.6.2 Search word

Let *self* be a Trie data structure that represents a vocabulary of words. Let *word* be the word we want to search for in the Trie.

The search word algorithm works by starting at the root node of the Trie and iterating through each letter in the word. At each letter, we follow the corresponding edge in the Trie to the next node, which corresponds to the next letter in the word. If we reach the end of the word and the node we are currently at has a True “term” variable, then the word is in the Trie. Otherwise, the word is not in the Trie.

Algorithm 12: Search Word using trie

Input: Trie data structure *self*, word to search *word*

Output: Boolean Whether the word is a valid word or not

```
1 root ← self
2 for letter in word do
3   if letter in root.alp then
4     root ← root.alp[letter];
5   else
6     return False;
7 return root.term;
```

5.6.2.1 Proof of correctness

At the end of the word, we check if the “term” variable of the current node is True. If it is, then the word is in the Trie and the algorithm correctly returns True. Otherwise, the word is not in the Trie and the algorithm correctly returns False. As the trie marks *term* as true for each valid word when inserting words in the vocabulary, it is ensured that any valid word will be found because of the standard implementation of trie and we will be able to successfully return True if the word is a valid word.

5.6.2.2 Time complexity analysis

The time complexity of the search word algorithm in a Trie is $O(n)$, where n is the length of the word being searched. This is because the algorithm performs a constant amount of work at each node of the Trie, and there are at most n nodes to visit, one for each letter in the word. Specifically, at each node, the algorithm performs a constant amount of work to check whether the current letter is in the node’s children, and then follows the corresponding edge to the next node. If the word is not in the Trie, the algorithm will visit all n nodes in the worst case, and therefore its time complexity is $O(n)$.

5.7 Brute Force

Let V be a set of valid words, and let *word* be a word we want to check if it is in V . A brute force algorithm for checking if *word* is a valid word involves iterating through each word w in V and checking if w is equal to *word* by iterating through each letter in both words and comparing them. If all letters in w match the corresponding letters in *word*, then *word* is equal to w and thus a valid word. Otherwise, we move on to the next word in V and repeat the process. If we reach the end of V and have not found a match for *word*, then *word* is not a valid word.

Therefore, by iterating through each word in a set of valid words and then iterating through each letter in each word, the brute force algorithm correctly determines whether a word is a valid word or not. However, the time complexity of this brute force algorithm is $O(nm)$, where m is the number of words in V and n is the maximum length of a word in V . As a result, this brute force algorithm is not efficient for large sets of valid words. But compared to the trie algorithm, which calls create trie only once and then search word finds the word in $O(n)$, this brute force algorithm takes $O(mn)$ for each word validity check.

Algorithm 13: Check Valid Word Brute Force

Input: List of valid words *corpus*, guess word *guessWord*

Output: Boolean Whether the guess word is valid or not

```
1 validWord  $\leftarrow$  False
2 for word in corpus do
3   if word == guessWord then
4     validWord  $\leftarrow$  True
5     break
6 return validWord;
```

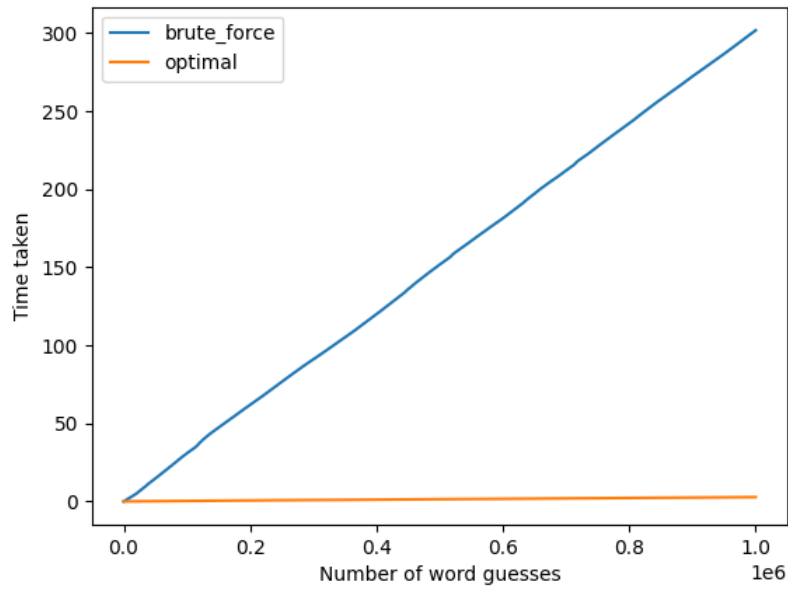


Figure 12: Caption

Graph comparing the two algorithms for 1000000 iterations is shown in the [12](#). This clearly shows the optimization achieved by using the trie data structure. Brute force method is linear in the number of words, whereas trie approach depends only on the length of the words we are using which in our case is 5 (as is allowed in wordle game). So it is $O(1)$.

6 Problem 4: Evaluate guess word

After checking the validity of the word, the evaluation bot should update the board state by returning the integer corresponding to the color indicators for each letter of the guessed word.

6.1 Instance

The algorithm received the current guess w_k and let $l_k = l_{k1}, l_{k2}, ..l_{kn}$ denotes the letters of the word w_k . For each l_{ki} in l_k , the algorithm should output an integer corresponding to 1(GRAY), 2(YELLOW), or 3(GREEN) to denote the color indicator of that letter.

6.2 Solution Format

A string res of length n consisting of 1, 2 and 3, where n is the length of the word.

6.3 Solution Constraints

- The length of res should be equal to n
- For every character r_i in res $r_i \in [0, 1, 2]$

6.4 Solution Objective

To find the colors corresponding to each letter of the guessed word by comparing it with the correct word.

6.5 Algorithms

The algorithm takes in the guess word and correct word as input, and returns a list of colours for each letter in the guess word, as well as a boolean value indicating successful execution. The algorithm initializes an empty list for the letter colours, and converts the guess and correct words to lowercase for easier comparison. It then loops through each letter in the guess word, and appends the appropriate colour to the list based on the rules specified -

1. If letter not present in any position of the correct word, the letter will be flagged as GRAY.
2. If letter present in some position of the correct word but not in the position guessed, the letter will be flagged as YELLOW.
3. If the letter is present in the same position as the correct word, it will be flagged as GREEN.

Algorithm 14: GetLetterColours

Input: Guess word *guessWord*, correct word *correctWord*

Output: List of colours for each letter and boolean value indicating successful execution

```
1 Initialize an empty list letterColours;  
2 guessWord  $\leftarrow$  guessWord.lower()  
3 correctWord  $\leftarrow$  correctWord.lower()  
4 for ind, letter in enumerate(guessWord) do  
5     if letter not in correctWord then  
6          $\lfloor$  letterColours.append(Colour.GRAY.value);  
7     else if letter == correctWord[ind] then  
8          $\lfloor$  letterColours.append(Colour.GREEN.value);  
9     else  
10         $\lfloor$  letterColours.append(Colour.YELLOW.value);  
11 return letterColours, True;
```

6.6 Time complexity analysis

As we are iterating through each letter of the guessed word and checking if the letter is present in any position, the time complexity is $O(n)$, where n is the length of the word, which we have fixed to a constant 5.

7 Model Extensions

Our current implementation can be generalized to varying word size, the number of attempts, and the vocabulary size. Our code details will be shared in the next submission.