Wordle Solver

An investigation undertaken by Richard Pawson

Documented as a possible NEA Project submission for the AQA A-level Computer Science qualification

January 2023

Introduction

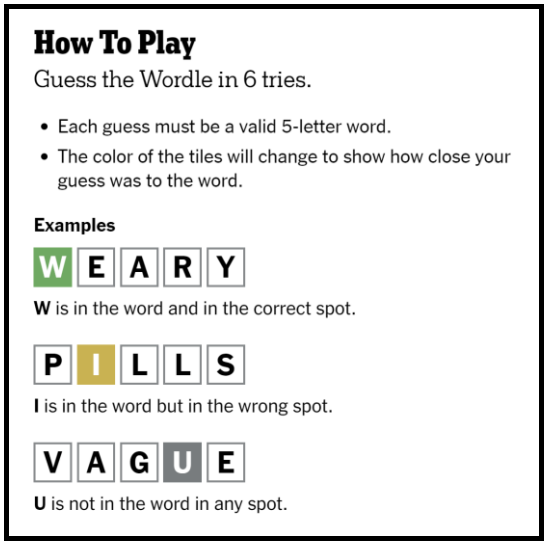
Wordle is a word puzzle in which the player attempts to guess a hidden five-letter 'target' word in a maximum of six attempts, receiving feedback on each attempt. It is similar to several older pen-and-paper puzzles. Wordle was invented by software engineer Josh Wardle, originally for his own personal use, but was made public in late 2021. There are now many copies and variants available online but the original version – now owned by the New York Times – offers a new puzzle each day, and has attracted an estimated 3 million players worldwide.[1]

The overall aim of this project is to design and develop a fully-automated Wordle 'solver'. A quick search on Google makes clear that such solvers have already been built, just as software developers have created automated solutions to most board games and puzzles. Although aware of the existence of automated solvers, however, I have not looked at any of them, because my motivation is to explore for myself, working from first principles, how to design an automated solution.

[1] https://www.nytimes.com/games/wordle

Analysis

My Wordle Solver will be designed specifically to work with the official version of Wordle. It might be adaptable to work with many of the variants, but those are not within the scope of the project. The screenshot below, captured from the official website, explains the rules:

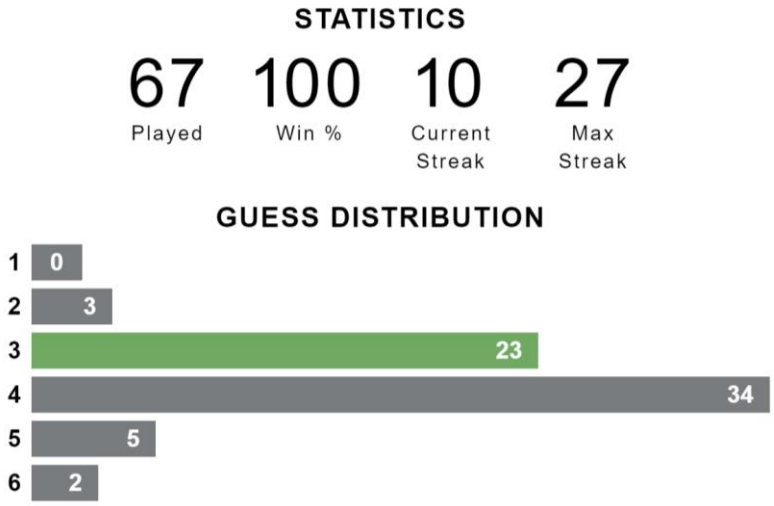


Of critical importance is the statement 'Each guess must be a valid 5-letter word'. As with the board game Scrabble, human players with higher vocabulary have an advantage, but unlike with the official rules of Scrabble, in Wordle there is no rule against looking up words. Also, unlike Scrabble, if you were to try a combination of letters that is not a valid word (for example **AEIOU**) there is no penalty – the system just advises you that your attempt is **Not in word list** and the attempt is not scored and not counted.

Wordle's publisher does not specify the list of valid words, but lists of valid Wordle attempt words are widely available online. *Many* of these attempt words are obscure. For this project, I have used a list of 12,947 such valid words, and which the source claims are all valid for use as Wordle attempts (see Appendix I: Lists of valid words and possible answer words).

However, it is not the case that the daily 'target' word can also be any valid word. The official site says nothing about this, but it can be determined from the JavaScript source code that the daily word is chosen from a rather smaller list of 2309 words (see, again, Appendix I: Lists of valid words and possible answer words). This makes sense: it avoids the inevitable howls of protest if the day's target word were, say, **XYLIC** or **WEKAS**, which are probably not widely known. The 2309 **possible answers** have obviously been chosen on the basis that they are likely to be familiar to a large section of the English-speaking populace – although they do include some American spellings (e.g. **VALOR**) and idioms (e.g. **HOWDY**). Surprisingly, many common 5-letter words do not appear in this second list: in particular, plurals of common 4-letter words such as **NAILS** and four-letter verbs with a **D** added for past tense such as **ACHED**. Presumably this decision has been taken to maximise variety.

When each daily puzzle is finished – either because the user has identified the word or has used up six attempts without identifying it – the result is added to a tally of the user's previous scores. The screenshot below shows my own personal stats from playing Wordle online (without help from any program), as of December 7th 2022.



There are many articles online about Wordle performance, including one[2] which offers the following 'key findings':

- *Canberra, Australia is the global city with the best Wordle average: 3.58 guesses.*
- *Sweden is the world's best country at Wordle, with an average of 3.72.*
- *The US is ranked #18 in the world for Wordle, with a national average of 3.92.*
- *The American state with the best Wordle average is North Dakota (3.65).*
- *The US city with the best Wordle scores is Saint Paul, Minnesota, with an average of 3.51.*

Unfortunately, this 'information' needs to be taken with a large pinch of salt! The official Wordle site does not disclose *any* statistics. Surveys like the one above are based on mining Twitter, Facebook and other social media for users' postings of their own scores. You don't need to be a professional data scientist to realise that this is unreliable data: people are more likely to publish their Wordle scores when they are performing well than when they are performing badly! Furthermore there is plenty of scope for enhancing your score: just play it once through on one computing device, then play it again on a different device, now knowing the answer, but still taking two or three faked guesses to lend credibility to your claimed scores.

[2] https://word.tips/wordle-wizards/

Player interviews

In the course of this project, I interacted with more than a dozen family members, friends, and colleagescolleagues, who play Wordle regularly, to try to ascertain how they go about trying to solve a Wordle puzzle, and how well their approach works. Initially, these discussions were informal and unstructured; later I added more structure in the form of fixed questions asked face-to-face or (in most cases) by email; some of those were followed up with further unstructured interaction. The following is what I learned from those interviews:

- All the individuals I interviewed had played Wordle regularly for between 2 and 20 months.

- When asked what percentage of the daily puzzles they successfully solved within six attempts, the responses all ranged between 97% and 100%. However, despite being asked, *only one person provided their actual statistics from Wordle*. Several said that they did not have the statistics for whatever reason, for example because they used different machines. I have no reason to doubt their truthfulness, but one does wonder how they know a figure like, say, 98% so accurately if they don't have the official statistics. It is also possible that their claimed figure accurately represents their more recent results, but they preferred not to disclose the full statistics.

- When asked for their average number of guesses, most people responded either with a specific number between 3 or 4 (3.5) or more vaguely 'between 3 and 4'. The official Wordle statistics page does not show your average score. When I gently probed a few of the respondents (I did not want to insult their intelligence) I established that several did not know how to correctly calculate the average. For my own statistics (shown on the previous page) the average is calculated as (0x1 + 3x2 +23x3 + 34x4 + 5x5 + 2x6)/67 = 3.7. I believe that several were confusing the idea of average (or mean) with 'mode' – the location of the highest point on the distribution curve.

- When asked if they had a preferred word (or words) for the first attempt, all affirmed that they did. Specific words cited included (listed in alphabetic order): **AMIGO, ADIEU, ALERT, ALTER, ANGEL, ARIEL, ATONE, AUDIT, CLOUD, DREAM, FEAST, GREAT, HOTEL, IDEAL, PEARL, RADIO, TREAD**. Those that said they had more than one first attempt word, said this was to add variety – there was (unsurprisingly) no reason given for which one they would choose at the start of any puzzle. It is clear that most people favour words with at least two vowels, and in many cases three.

- When asked if they had a preferred word (or words) to use as the second attempt if their first attempt came back with no matches, most had. Typically the second attempt used as many as possible of the vowels not used in the first word, and all different consonants. Examples given included (in alphabetical order): **AUDIT, CLOUD, COULD, COUNT, HOTEL, MOUND, PITHY, PLUCK, SCOUT, SHOUT, SOUND**. One respondent effectively had pairs of words with no common letters, where they used either of the pair as the first attempt, and the other as the second *if the first produced no matches.*

- Next I asked if they believed that *each attempt word* should be a possibility for the (hidden) target. For example each letter already marked as green should occur in the same position in each subsequent attempt word, and each yellow letter should also occur and in a *different*

position. The majority of respondents confirmed that they adopt this approach. This was my first surprise from the interviews, because I have never consciously used that approach. For example, even if my first attempt has scored one yellow and one green, for my second attempt I will often use a word that has *all* different letters – with the aim of gaining more information. This means that I consciously forgo the possibility that I might 'get it in two'. In follow-up discussions I found that several players *in both camps* felt quite strongly that theirs was the better way to play.

My second surprise in regard to this question was when one respondent said that the first approach was necessary in order to play Wordle in 'Hard mode'. Until that point I had not heard of 'Hard mode'. It is not mentioned in the rules, and its existence is not readily apparent from the user interface (you have to explore **Settings** in the top-right corner). Hard mode enforces the rule that every attempt word *must* be consistent with the marks from previous attempts. But when I followed up on my question (with a few respondents) I found that most, like me, were not aware of Hard mode. This indicated that most people who were adopting this principle, were doing so because they believed that was the best way to solve the puzzle, even in the normal (not Hard) mode. They said things like, 'You should never pass up the possibility of getting the target word with your next attempt.'

- I finished the structured part of the interaction with questions designed to glean how well the players knew the extent of the difference between the number of words that may be used as attempts, and words that can actually be target words. This showed that only around half of respondents knew that there was any difference. For those that did not I ignored their answers to the next two questions – and in fact several some admitted that they did not really understand them.

- I asked: "Which of the following 'words' do you think would you be allowed to use in as an attempt word: **AALII**, **POGAL**, **QUOPH**, **SMAAK**, **XYSTS**, **ZURFS**?" Almost all responded that none were allowed words; one thought that **POGAL** might be allowed. In fact they are all allowed attempt words *except* **POGAL** and **QUOPH**, which I had made up – as a control. It is hardly surprising that most people did not know that **AALII**, **SMAAK**, **XYSTS**, **ZURFS** are valid attempt words. I would not have known either except that I had access to list of more than 12,000 acceptable 5-letter attempt words.

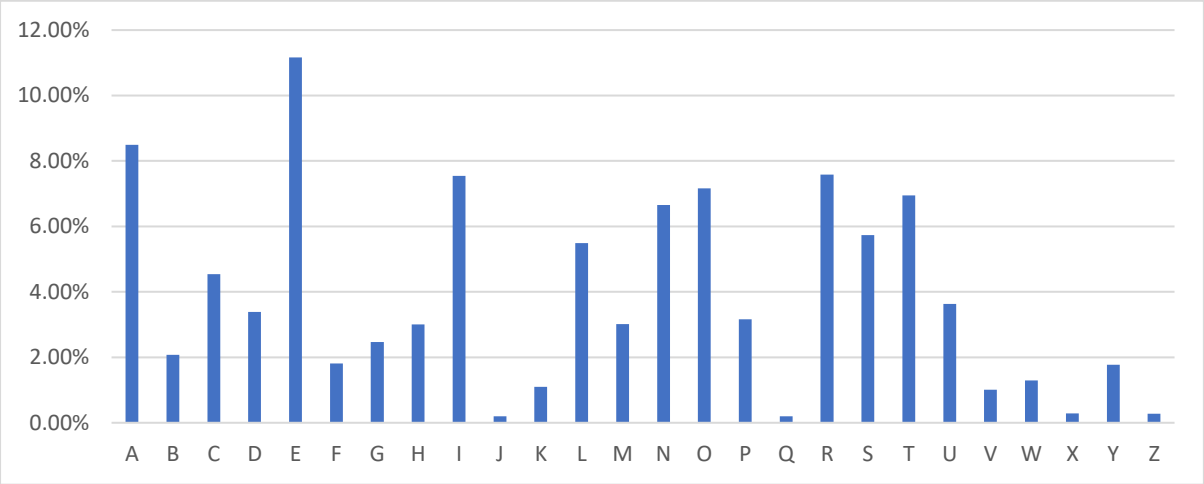
- Finally I asked "Which of the following words do you think could be actual 'target' words set in the official Wordle game: **ACHED**, **BUTTE**, **JUNTO**, **MYRRH**, **VIGOR**, **ZONES**?" Some thought that **BUTTE** or **JUNTO** might not be valid target words (they are). The surprising thing was that most thought that **ACHED** and **ZONES** were valid target words, when in fact they are not. They were unaware of the fact that plurals ending in **S** and past tenses ending in **D** are never set as target words.

In summary, the conclusion from this analysis of player interview is not that the players I know are ignorant – it is that an automated computer solver could use a vastly larger number of attempt words than the typical human player, and that there are at least some rules of thumb that it could apply which appear unknown to many *competent* players. This gave me some encouragement in my quest. It was now time to start thinking about possible algorithms.

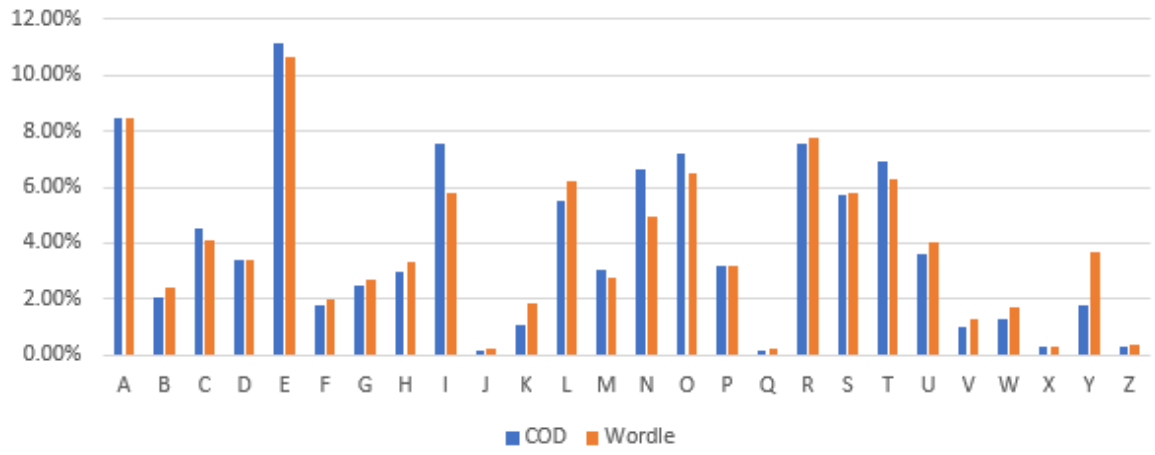
Initial modelling

It appears – both from my interviews and from reading many blogs about Wordle playing strategy (of extremely variable quality!) – that most players are, at least to some extent, seeking to use words that cover the most frequently occurring of the yet to be identified letters. This could certainly form the basis for an automated Wordle solving algorithm

The graph below shows the letter frequency for all words in the Concise Oxford Dictionary (9th edition, 1995).[3]



As a first step I decided to analyse the letter frequency specifically in the 2309 **possible answer** words for the official Wordle game and compare this to the previous analysis:



The distribution of letter frequency is *broadly* similar, though there are notable differences for the letters **I**, **K**, **N** and, especially, **Y**.

It is also fairly obvious that the letter frequency is not the same for all positions. So I next generated a frequency analysis of letters for each of the five letter positions in the 12,947 **valid words**. I wrote this as a single function using 'expression syntax' (the function returns a single expression):

```
static IEnumerable<(char,int)> CharacterCount(IEnumerable<string> words, int charNo) =>
words.GroupBy(w => w.ToCharArray()[charNo]).Select(g => (g.Key,g.Count())).OrderBy(t =>
t.Item1);
```

[3] https://www3.nd.edu/~busiforc/handouts/cryptography/letterfrequencies.html

This function was called five times with the value of `charNo` ranging from 0 to 4. The results are shown in the following table. For each column, the letters are presented in relative order of frequency *in that position*:

1		2		3		4		5	
S	1560	A	2260	A	1235	E	2323	S	3950
C	920	O	2093	R	1197	A	1073	E	1519
B	908	E	1626	I	1047	T	897	Y	1297
P	857	I	1380	O	989	I	880	D	822
T	815	U	1185	N	962	N	786	T	726
A	736	R	940	E	882	L	771	A	679
M	693	L	697	L	848	R	716	R	673
D	681	H	544	U	666	O	696	N	530
G	637	N	345	T	615	S	515	L	475
R	628	Y	267	S	531	K	500	O	388
F	595	T	239	M	510	D	471	H	367
L	575	P	228	C	392	G	422	I	280
H	488	M	188	D	390	P	418	K	257
W	411	C	176	P	363	C	406	M	182
K	375	W	163	G	362	M	402	P	147
N	325	K	95	B	334	U	401	G	143
E	303	S	93	W	271	B	242	C	127
O	262	D	84	K	268	H	235	F	82
V	242	B	81	V	240	F	233	X	70
J	202	G	75	Y	213	V	155	U	67
U	189	X	57	F	178	W	128	W	64
Y	181	V	52	Z	142	Z	126	B	59
I	165	Z	29	X	133	Y	108	Z	32
Z	105	F	24	H	120	J	29	Q	4
Q	78	Q	15	J	46	X	12	V	4
X	16	J	11	Q	13	Q	2	J	3

Next, I repeated the exercise for just the 2,309 **possible answer** words. While they are similar, there are a few striking differences, particularly the ranking of **R**, in columns 2, 3, and 5 and the rankings of **S** and **D** in column 5. The last two are consistent with the earlier observation that the Wordle answer list does not include plurals ending in **S** nor past tense verbs ending in **D**. These differences are indicated by highlights in both tables:

1	
S	365
C	198
B	173
T	149
P	141
A	140
F	135
G	115
D	111
M	107
R	105
L	87
W	82
E	72
H	69
V	43
O	41
N	37
I	34
U	33
Q	23
J	20
K	20
Y	6
Z	3
X	0

2	
A	304
O	279
R	267
E	241
I	201
L	200
U	185
H	144
N	87
T	77
P	61
W	44
C	40
M	38
Y	22
D	20
B	16
S	16
V	15
X	14
G	11
K	10
F	8
Q	5
J	2
Z	2

3	
A	306
I	266
O	243
E	177
U	165
R	163
N	137
L	112
T	111
S	80
D	75
G	67
M	61
P	57
B	56
C	56
V	49
Y	29
W	26
F	25
K	12
X	12
Z	11
H	9
J	3
Q	1

4	
E	318
N	182
S	171
A	162
L	162
I	158
C	150
R	150
T	139
O	132
U	82
G	76
D	69
M	68
K	55
P	50
V	45
F	35
H	28
W	25
B	24
Z	20
X	3
Y	3
J	2
Q	0

5	
E	422
Y	364
T	253
R	212
L	155
H	137
N	130
D	118
K	113
A	63
O	58
P	56
M	42
G	41
S	36
C	31
F	26
W	17
B	11
I	11
X	8
Z	4
U	1
J	0
Q	0
V	0

The second table almost invites you to identify possible start words that might have a high chance of scoring at least one Green. **SAINT**, **CRANE**, **SHINE**, should be good candidates on this basis (none of which were given by any of the interviewees). However, it is not possible to determine – just from the data shown above – the actual probability that any of these words would deliver a green, since the entries in each column are *not* independent of each other. (That is because there are only 2309 **possible answer** words, not 26^5). This hints at some of the problems of basing an automated solution on letter frequencies.

Beyond the first, and in some cases, second attempt, *none* of the players I interviewed could articulate the process by which they choose their subsequent attempt words. It is clear that many are using a large mental 'rule base' relating to word structure. Some of these rules may have been taught explicitly in early schooling ('*I before E except after C*', for example); but dozens, possibly hundreds of other rules are just learned unconsciously over many years of English usage. Consider the following real example from tackling Wordle 473 (5 Oct 2022):

The word has scored one yellow and two greens. But without needing to run through a physical, or mental, dictionary of all possible answers, a human player just *knows* that there is no possible word ending in **SR**, so *effectively* they already have three greens:

Whether they are conscious using heuristics or not, many human players would then deduce – still without considering specific possible words – that the last letter (if it cannot now be **E**) is most likely to be a **Y**, a **T**, or an **H**, and that the first letter is a consonant. It is clear that human players are considering far more than the simple letter frequency shown earlier (or an approximation thereof). They are considering the pairing or grouping of letters, and/or their specific position within the word.

Whether a player consciously adopts the 'every attempt should be a possible answer' rule or not, it is clear to me that from the third attempt most players are always selecting a word that *could be* the target. This might be because the number of remaining possible answers is reducing each time so the chance of getting the answer at the next attempt increases. It might also be it can be difficult for a human to identify any word that matches all the constraints – so that when you do identify a matching word there may be a strong urge to try it out, without continuing the search. And what would be the point of continuing the search, some players argue, when any matching word is as likely as any other?

Nonetheless, human beings are astonishingly good at some forms of pattern matching. Crossword enthusiasts often report that – given a few letters in their correct places – they have a strong sense of 'seeing' the word. (It is perhaps less easy in Wordle, where the known letters are not always in the correct place. Though, again, experienced crossword players can be good at 'just seeing' anagrams.) *Aural* memory also plays a role: given the ending of a word, *including a vowel* sound, we find it easy to list words that rhyme with it; not all the rhyming words will be an exact letter match, but it is still a powerful capability when playing Wordle. Similarly, we can easily list words that start with the same consonant-then-vowel sound. Going back to the example above, my aural memory does not recall a common short word that sounds like **_ARSY**, or **_ARST**, so that leaves **_ARSH**, from which my rhyming neuron suggests: **HARSH**, or **MARSH**.

Potential strategies for an automated Wordle solver

If the mental rule base of a good human player *could* be captured, then it would be possible to build a kind of 'expert system' from it, either using a ready-made 'inference engine' or built as a bespoke program. For the latter, the programming language Prolog would be an obvious choice, being a 'declarative' language where the programmer specifies the rules/transformations but leaves the decision on how/when to apply them to the language's run-time inference engine.

The challenge for this approach – as is often the case with expert systems – would be identifying the rules. This challenge has encouraged the growth of 'machine learning' techniques. In *theory,* given a large enough set of examples of actual Wordle games (i.e. the record of attempts and their outcomes) it should be possible to apply a machine learning algorithm either to deduce the rules explicitly for itself, or model them implicitly. The latter approach includes the use of a neural network – where the resulting set of synaptic weights could execute the strategy (even for words not previous seen in the training examples) without being able to articulate any rules.

The chosen approach

The approach adopted for this project, however, does not use any of what would today be classed as 'AI' or 'machine learning' techniques. (Though, had this project been conducted 40 years ago it would likely have been classified as 'AI' just given the nature of the problem being tackled. As Marvin Minsky, one of the founding fathers of AI has been quoted as saying, '"AI" is the name we give to all that stuff we don't know how to do yet.')

The chosen approach for this project uses 'plain old programming', and in a mainstream programming language: C#. My objective is to use only the constructs that come with the C# 9 language implementation (released in 2020) - *with no dependence on any additional framework, library, or package*. The reason for this choice is that I want to explore an algorithm that has no need for fancier techniques, based on the following insight.

Most human players are – consciously or unconsciously, and whether playing in regular mode or Hard – seeking to identify the most likely *answer word*. But a completely different strategy could be: for each attempt i*ncluding the first*, just seek to *eliminate* as many words as possible, eventually reducing the possible answers to one.

There are *some* circumstances where it may readily be seen that this alternative (I will call it 'elimination') strategy offers advantage. Consider the following game:

The player has had very good luck: getting four greens on the first attempt. There are limited possibilities for the first letter. With luck the player will get the answer on the next attempt, and surely it will be solved within six? Here's how the game played out, though:

Fail! The answer was the only other possibility: **WIGHT**. Following the 'elimination' strategy, however, the computer would identify the six remaining possible answers and pick a word that eliminated as many as possible. **SWARM**, for example, though not a possible answer itself, would reduce the remaining possibilities to two: either a yellow letter, or green **S**, would indicate the first letter of the answer, or all-greys would indicate either **FIGHT** or **LIGHT**. Using the elimination strategy means that the player foregoes the 1/6th probability of getting the answer (to this specific puzzle) on the second attempt, but is instead *guaranteed* to get it on the third or fourth.

This project will apply the 'elimination' strategy from the outset. For the each attempt it will evaluate each of the 12,947 valid words to identify which one eliminates the remaining **possible answers**. It will only ever use a possible answer as the attempt if that word happens to be the best word for reducing possibilities (or at least was one of several words laying equal claim to being the best). Eventually, this algorithm will reduce the remaining possibilities to one. The question is: in how many attempts? After the mark for each attempt is given, it will determine how many of those starting **possible answers** are still 'in the running' and start the next cycle with that.

The overall algorithm is a version of a well-known algorithm in computer science known as 'minimax', which is widely used in games, decision theory, statistics, and philosophy[4].

How is the computer to determine the 'best' of the 12,947 words in each case, given that any attempt can have many possible outcomes? In theory, given that an outcome for a given attempt shows each of five letters as either Green, Yellow, or Grey, there are 243 (3^5) possible outcomes. However, an outcome that shows four greens and one yellow is impossible – if the position of four letters is known, the position of the fifth cannot be unknown. That leaves 238 theoretically possible outcomes. As a game progresses the number of possible outcomes will steadily reduce.

In assessing each of the 12,974 **valid words** for the next attempt, we need to calculate the distribution of possible outcomes for that word. We are not interested in the *best possible* outcome: because best possible outcome will always be the word that just happens to be the correct answer. Rather, we are interested in narrowing the field in one of two ways:

1) Selecting the word where the *worst possible outcome* for that word would leave the smallest number of remaining possible answers.
2) Selecting the word where the (mathermatically) *expected* number of remaining possible answer after the outcome would be the smallest.

In the second variant the mathematical concept of 'expected value' is defined as:

$$\Sigma\ p(o).n(o)$$

which may be read as: for each possible outcome **o**, the sum of the probability of getting that outcome – **p(o**) – multiplied by the number of words remaining from that outcome – **n(o)**.

A hypothesis to explore is that these two variants might optimise for different goals:

- If the goal is to minimise the *maximum* number of attempts taken to guess a word, then perhaps the first definition might work best.
- If the goal is to minimise the *average* number of attempts taken to guess the word, then the second definition might work better.

Hard mode

The algorithm as stated will not work in Hard mode, which would not, for example, permit **SWARM** as the second attempt in the puzzle above. It is possible, however, that with a small modification it could work in Hard mode. This just requires that the search for the attempt word that would eliminate the most possibilities be restricted to the remaining possible answers. This possibility will also be explored.

[4] https://en.wikipedia.org/wiki/Minimax

Mock-up of the user interface

The popularity of the online Wordle game may – at least in part – be attributed to its user interface: clean, simple, and clear. Anyone coming new to Wordle, but who is already familiar with mobile phone apps, will likely figure out how to use it in seconds.

However, this project is not developing a system for such users: it is a scientific investigation, and the sole intended user is the project author. Investing time and effort in a graphical user interface like that of Wordle would add no benefit. Also, the role of the user in this project is *not* to view the patterns of outcomes and thence to guess the word – the computer will be doing that. Here, the user is simply there to give the feedback – to 'mark' the attempt words offered by the computer against the target word. (The user might have set target word and evaluate the marking themselves, or they might act as an intermediary between the live daily Wordle game and the solver program).

Even for a graphical user interface, designing a means to mark each letter of an attempt word as green, yellow, or grey cannot be made intuitive. Perhaps the user could select on each letter in turn, and for each one *then* click on one of three colours in a separate palette. Alternatively, clicking on a letter could turn it grey; clicking twice could turn it yellow; three times green; a fourth time cycling back to grey in case there is a need to correct an erroneous entry.

But why bother with any of this? All that is required is to enter five marks, corresponding to: an in-place match (Green on the Wordle app); an out-of-place match (Yellow on the Wordle App); or a no-match (Grey on the Wordle App). It would be simpler to use three non-alphabetic symbols, each entered as a single keystroke, for example:

***** for an in-place match
+ for an out-of-place match
_ for a no-match

And since the standard Console UI uses a monospace typeface, the symbols can be entered underneath the system-generated 'attempt' word, and they will neatly line up with the characters above, as shown in this sketch:

After each attempt is presented by the Solver, the user simply types in five symbols immediately underneath. Not pretty, but simple, effective, and efficient.

Adopting a functional approach to design and coding

Although technical implementation does not normally form part of the analysis, it is being included here because it is an *explicit objective of this project* to adopt the 'functional programming' (FP) approach – as far as is possible – within the C# language. The rationale for this is in part because I wish to expand my skills in FP – given that in professional development circles, FP is widely recognised as the next paradigm in programming, and its adoption is growing rapidly.

However, there is also a potential immediate benefit to this project. One of the claimed advantages of adopting FP is that it facilitates parallel execution – provided that the host computer has multiple processor cores. Since my laptop has four physical cores, I hope to demonstrate that the heavy processing that the algorithm requires can take advantage of that. (The four physical cores in my laptop are configured as eight 'virtual' cores. However, this further 2x split does not offer any advantage to the kind of intense processing that this program will require: the advantage of virtual cores is clearer when swapping frequently between different programs, not running one program as parallel threads.)

Adopting FP means that the core functionality must be built exclusively from 'pure' functions.

Definition of a 'pure' function

Many constructs in programming that are referred to as 'functions' are not pure functions. For a function to be pure:

- It must return a value (though this value may be a data structure).
- It must define parameters
- The returned value must be derived solely from the parameter values, deterministically. The function must not depend on any other information coming from the system, such as global variables, user input, random number generation, or the current date/time.
- It must not have any 'side effects': it must not make any changes outside the function, such as changing a global variable, persisting a value to a file or database, or even just writing to the screen. Changing any of the values or references passed in as a parameter would also be a side effect. The only permissible effect of the function that is visible from the outside, is the returned value.

User interface

In a pure FP programming language – such as Haskell – it is possible to implement a user interface from pure functions. However, this is not possible in C# and other 'mixed paradigm' languages. Therefore, the constraint of using FP approaches applies only to core functionality, not to the UI. Hopefully though, the code to implement the simple UI sketched earlier can be kept very small.

Coding the functions using functional programming idioms

A decision has also been taken not just to adopt an FP approach to the *design* of the core system, but to the *implementation* also. This means adopting pure FP patterns or 'idioms' for the coding itself. Specifically, the objective is that:

- All functions will be written using C#'s 'expression syntax' which requires that functions take the form:

  ```
  <ReturnType> FunctionName(<parameter definintions>) => <expression>;
  ```

  For example

  ```
  int Square(int x) => x * x;
  ```

  which is equivalent to:

  ```
  int Square(int x)
  {
      return x * x;
  }
  ```

  Adopting the expression syntax is terser, while being just as easy to read (once you are familiar with it). Moreover, it forces the writer to adopt other FP coding patterns because a function written with expression syntax cannot include a 'block' of code (a sequence of statements).

- Procedural-style loops (such as `for` and `while`) are not permitted in FP. Iteration can be achieved by making functions recursive. However, I have set the additional objective of writing no recursive functions, in order to show that all iteration on lists (and there will be plenty of that) could be implemented entirely through ready-made 'higher order functions' including 'map', 'filter' and 'fold' (also called 'reduce'). In C#, the 'map' concept is implemented as the (LINQ) `Select` function and 'filter' as `Where`. The generic capability 'fold/reduce' is provided in C# as the `Aggregate` function, but there are also more specific (and easier to use) versions of 'fold/reduce' including `Count`, `Sum`, `Average`, `Min`, `Max`. Other standard C# higher-order functions that may be of use include `OrderBy` (a flexible sorting capability) and `GroupBy`.

Summary of SMART objectives

Ten 'SMART' (Specific, Measurable, Achievable, Realistic, Timely) objectives have been identified for this project. Five are concerned with the functionality of the resulting system –the 'external' or 'what' perspective. The other five are concerned with the *implementation* of that functionality – the 'internal' or 'how' perspective, although for most projects the latter would not be considered appropriate as objectives. However, the 'functional programming' approach (as explained in the previous section) has been voluntarily adopted as an explicit objective – recognising that it would be possible to achieve the first five objectives using purely procedural or object-oriented approaches. The objectives are numbered here for reference only not to specify priority.

Objectives concerning functionality and usability of the resulting system

1. The system should be capable of solving the daily online Wordle puzzle, requiring the investigating user only to map the inputs and outputs between the two systems, not to provide any other assistance.

2. Prove that, across all possible target words for Wordle, the system will solve the puzzle in six or fewer attempts in at least 97% of cases (gauged – from the interviews - to be equivalent to a reasonably good human player).

3. Demonstrate the difference between adopting the 'worst case' and 'expected' alternative variants of the algorithm (in terms of percentage of puzzles solved in six attempts, and the average number of attempts taken, across all 2,309 possible target words).

4. Show that the algorithm could also solve puzzles in the 'Hard mode' setting, and determine the effectiveness relative to 2, using the same criteria as given in 3.

5. Demonstrate that the minimalist user interface sketched previously, while designed only for the author's use, could be used by an external validator following simple written instructions.

Objectives concerning the implementation

6. All core functionality (everything except the minimal user interface code) to be provided by 'pure' functions. (see Definition of a 'pure' function.)

7. All core functions to comprise a single statement returning the value of an expression.

8. All core functions to be implemented without need for recursive calls.

9. Demonstrate that the functional implementation can be parallelised for faster performance.

10. Provide 100% unit testing coverage for core functions.

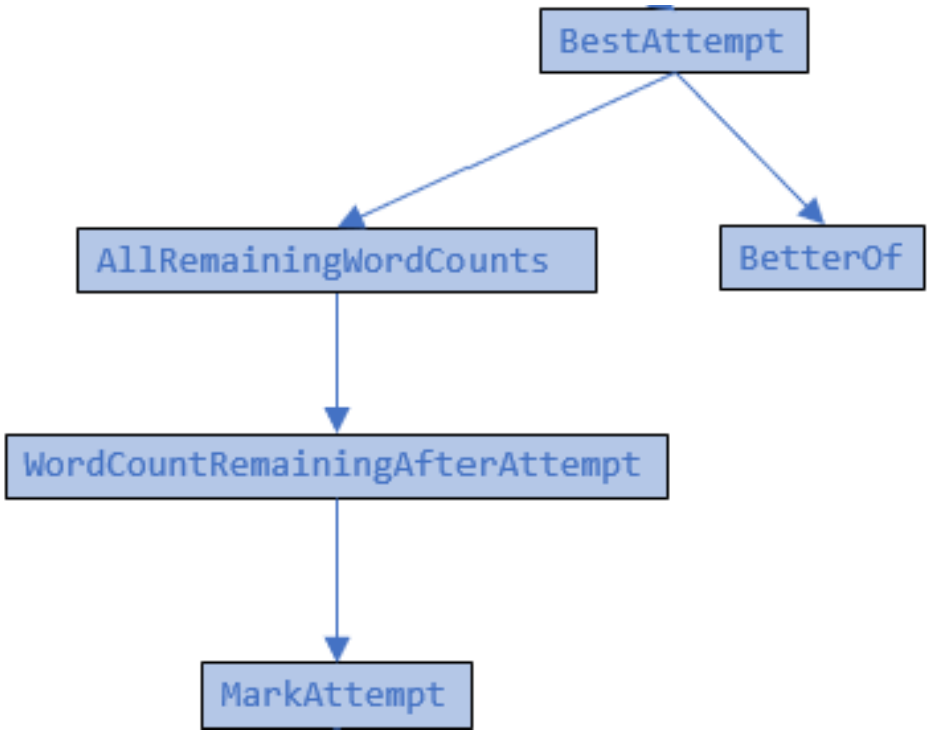
Design

Identifying the best word for the next attempt – algorithm and data structures

The following describes the proposed algorithm for finding the next 'best attempt' word, in a little more detail:

1. After each attempt, take the list of **possible answers** that remain valid, i.e. are consistent with the marks given for all attempts so far. (For the *first* attempt this will be the list of 2,309 possible answer words used in the official game of Wordle).
2. Evaluate each of the 12,947 **valid words** as a possible next attempt. For each one, evaluate it as an attempt against the hypothetical target of each one of those *remaining* possible answers and group them by the outcome (each 'outcome' being a specific pattern of five Green/Yellow/Grey marks).
3. For each of the outcomes resulting from the previous step (which will always be less than or equal to 238) determine how many possible answers that outcome *would* leave.
4. Determine the *worst* outcome for each of the 12,947 valid words if selected as the next attempt.
5. The selected attempt should be the word (out of 12,947) where its *worst* outcome would leave the *smallest* number of remaining possible answers.
6. If the calculation would result in several potential attempt words with *equal* outcome, then if any of them happens *also* to be in the list of remaining possible answers (see step 1 above) then pick one of those words – it does not matter which.

The adopted design involves five functions, as shown in the dependency hierarchy below:



The following table defines the new functions in more detail:

Function name	Arguments supplied	Returns
`BestAttempt`	All possible answers at the current state of the puzzle. All valid attempt words.	The single word from the valid attempt words that represents the best option.
`BetterOf`	Two attempt words being considered, each paired with its 'count' (as defined above). The possible answers at the current state of the puzzle (needed to help choose between two words with the same count – see point 6. above)	Returns the better of the two words, still paired with its count.
`AllRemainingWordCounts`	All possible answers at the current state of the puzzle. All valid attempt words.	Each attempt word paired with the *count* of possible answers that would remain if that attempt word was used and receive the worst possible outcome (mark).
`WordCountRemainingAfterAttempt`	List of possible answer words at the current state of the puzzle. Attempt word.	Return the number of possible answers that could remain after this attempt. (See note below about alternative implementations).
`MarkAttempt`	Attempt word. Target word.	5 character string with symbols *****,**+**,**_** indicating in-place match, out-of-place match, and no-match

Alternative variants of the algorithm

The two variants on the main algorithm proposed in the Analysis section require only different implementations of the function `WordCountRemainingAfterAttempt`. Nothing else needs to change. The two variants are:

1. Return the number of remaining possible answers resulting from the *worst* outcome (mark) given to the attempt word.
2. Return the (mathematically) *expected* word count remaining after the attempt – effectively the weighted average of all possible outcomes.

Marking an attempt word against a target – algorithm and data structures

`MarkAttempt` plays a critical role in the implementation of this algorithm. It may be called many thousands of times in the identification of the next best attempt. Surprisingly, designing this function was much harder than I expected. The design outlined below seems quite simple and elegant now – but it evolved from a lot of much more complex and ugly attempts!

The official rules of Wordle are expressed concisely, but it turns out that marking an attempt is not as simple as it sounds. The complexity arises principally from repeated letters – either repeated in the target word, or in the attempt word, or both. If we evaluate the five letters of the attempt word one at a time, then we need to make sure that the evaluations do not trample over each other, i.e. do not count a letter in the target word more than once.

It is helpful to consider how a human might mark an attempt:

Target word: **FOLLY**

Attempt word: **FLOOD**

An effective approach is to start by identifying the in-place matches ('Greens'). Working through the attempt word one letter at a time, if it matches the corresponding letter in the target, the letter in the attempt word is marked green, and, importantly, the corresponding letter in the target is somehow also marked (shown as red below) to indicate that it has been 'used'. This is so that the latter in the target word is not accidentally matched again when evaluated for yellows:

Target word: **FOLLY**

Attempt word: **FLOOD**

Now considering the yellows, for each letter in the attempt word (we can skip those letters already marked green) we need to see if this letter can be found in the letters of the target word, *not already excluded* (red in the notation above). The **L** in **FLOOD** in has a match (actually two) in **OLLY**. So we mark the letter attempt word as yellow, but must also strike off one **L** (it doesn't matter which because all greens have already been processed) from the target word, *so that that same letter can't be counted twice*:

Target word: **FOLLY**

Attempt word: **FLOOD**

Similarly, we mark the first **O** in **FLOOD** as yellow and strike out the **O** from **FOLLY**.

Target word: **FOLLY**

Attempt word: **FLOOD**

The importance of this striking-out becomes more obvious when we come to the *fourth* letter (the second **O**). Because the **O** in **FOLLY** has already been used to match the first **O**, there is no remaining (uncoloured) **O** in **FOLLY**, so the second **O** in **FLOOD** is marked grey and not yellow. After then marking the **D** as grey, we end up with:

Target word: **FOLLY**

Attempt word: **FLOOD**

The above description would translate quite naturally into *procedural* code. But given the self-imposed constraints of FP, we need to find a way to express this in strictly functional terms. Consider even the idea of 'evaluate the greens *then* evaluate the yellows': in procedural coding that suggests *sequential* function calls to, say, `EvaluateGreens` and `EvaluateYellows`. In FP, this *could* be achieved by holding those two functions in a list and then iterating through that list of functions using, say, the 'fold' function. However, a simpler option in this case is just to 'compose' the two functions:

```
EvaluateYellows ● EvaluateGreens
```

C# doesn't have a composition operator (like ● above). It is possible to write one[5] but unless it turns out that this will be needed in several places, it is easier for one case simply to write something like this:

```
EvaluateYellows(EvaluateGreens(…))
```

In other words, call `EvaluateYellows` using the result of calling `EvaluateGreens` with the appropriate arguments (…).

We also need to think about the way to represent the arguments and the return value of these functions. It is clear that `EvaluateGreens`, at least, must pass back *both* the updated attempt word and the updated target word, which suggests a 2-tuple. What about the colours – green, yellow, and grey for the attempt, and our chosen colour of red for the letters in the target that are now used up (or 'struck out')? It would be possible to design a data structure that held five letters and a colour for each, but this is unnecessary. Once a letter *position* is marked up in a colour, we do not need to know its letter value anymore. So we could simply replace the letter with a non-alphabetic symbols instead.

Given that in the sketch of the console UI (see Analysis - Mock-up of the user interface) we have already decided on the symbols *****, **+**, **_** to allow the user to specify an in-place, out-of-place, and no-match mark respectively, why not use those same symbols in the core functional logic also? They are clear to read (by the programmer in this case) and this solution avoids unnecessary 'mapping' between the internal representation and the public (user) presentation. We might even call this the 'Richard Rogers school of architecture' since his buildings famously expose the services and support structure on the outside.We just need one additional symbol to represent what we have been showing as 'red' in the target word, so:

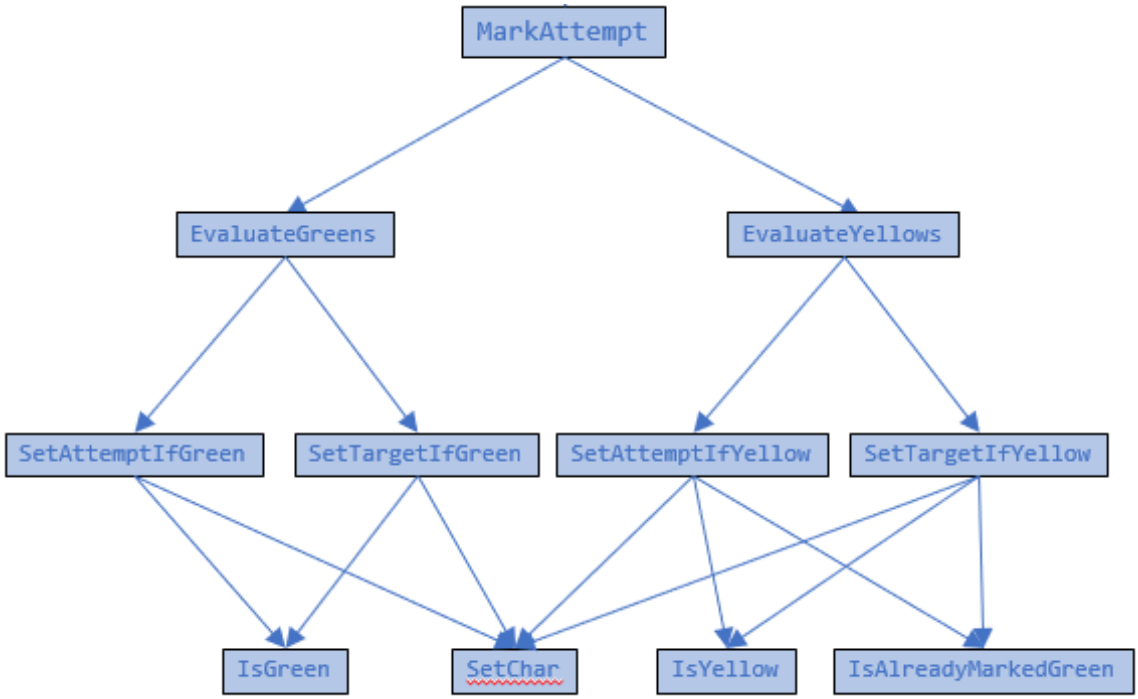
***** for green
+ for yellow
_ for grey
. for red

Returning to the previous example, `EvaluateGreens` would take `"FLOOD"` and `"FOLLY"` as arguments and return the tuple `("*LOOD", ".OLLY")`. These two values would need to be passed into `EvaluateYellows` which would return `"*++__"`.

[5] https://stackoverflow.com/questions/5264060/does-c-sharp-support-function-composition

Functional decomposition of the full marking algorithm

We have already seen that the `MarkAttempt` function will delegate to the functions `EvaluateGreens` and `EvaluateYellows`. The diagram below shows a complete functional decomposition:

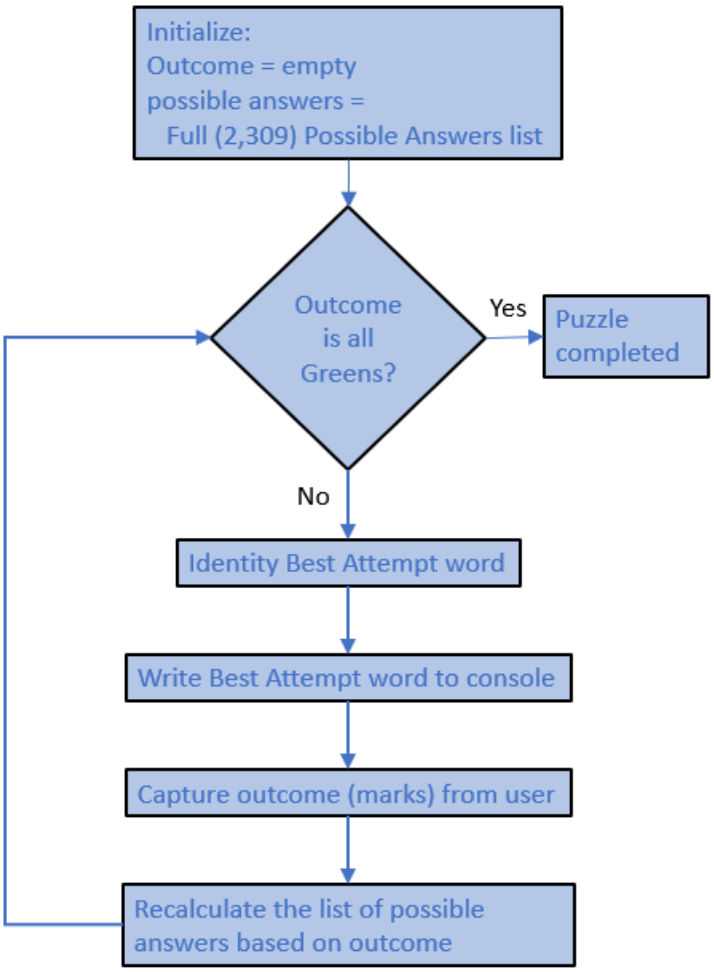


The following table specifies what is required of each of the functions shown above, from a purely external perspective:

Function name	Arguments supplied	Returns
MarkAttempt	attempt word target word	5 character string with symbols *,+,_ indicating in-place match, out-of-place match, and no-match
EvaluateGreens	attempt target	2-tuple comprising: attempt with all in-place matches replaced by symbol * target with all in-place matches replaced by symbol .
EvaluateYellows	attempt, assumed to be already marked up by EvaluateGreens. target, assumed to be already already marked up by EvaluateGreens.	2-tuple comprising: Input attempt word with all out-of-place matches replaced by symbol + Input target word with all out-of-place matches replaced by symbol .
SetAttemptIfGreen	attempt. target. charNo character number.	attempt with character replaced by symbol * if it is an in-place match with target
SetTargetIfGreen	attempt word. target word. charNo character number.	target with character replaced by symbol . if it is an in-place match with attempt
SetAttemptIfYellow	attempt and target, both assumed to have any in-place matches, and any *previous* out-of-place already marked. charNo character number.	attempt word with character replaced by symbol + if it is an out-of-place match with target
SetTargetIfYellow	attempt and target, both assumed to have any in-place matches, and any *previous* out-of-place already marked. Character number.	target with character number replaced by symbol . if it is an out-of-place match with attempt
IsAlreadyMarkedGreen	Attempt charNo	true if the attempt word has the * character in the specified character number
IsGreen	attempt target charNo.	true if the specified character number holds the same letter in both words
IsYellow	attempt targetcharNo	true if the specified character number of the attempt is contained in the *unused* letters of the target
SetChar	word (any5-letter word) charNo newChar, new character.	Input word with the specified character number replaced by the new character

Console user interface program - flow

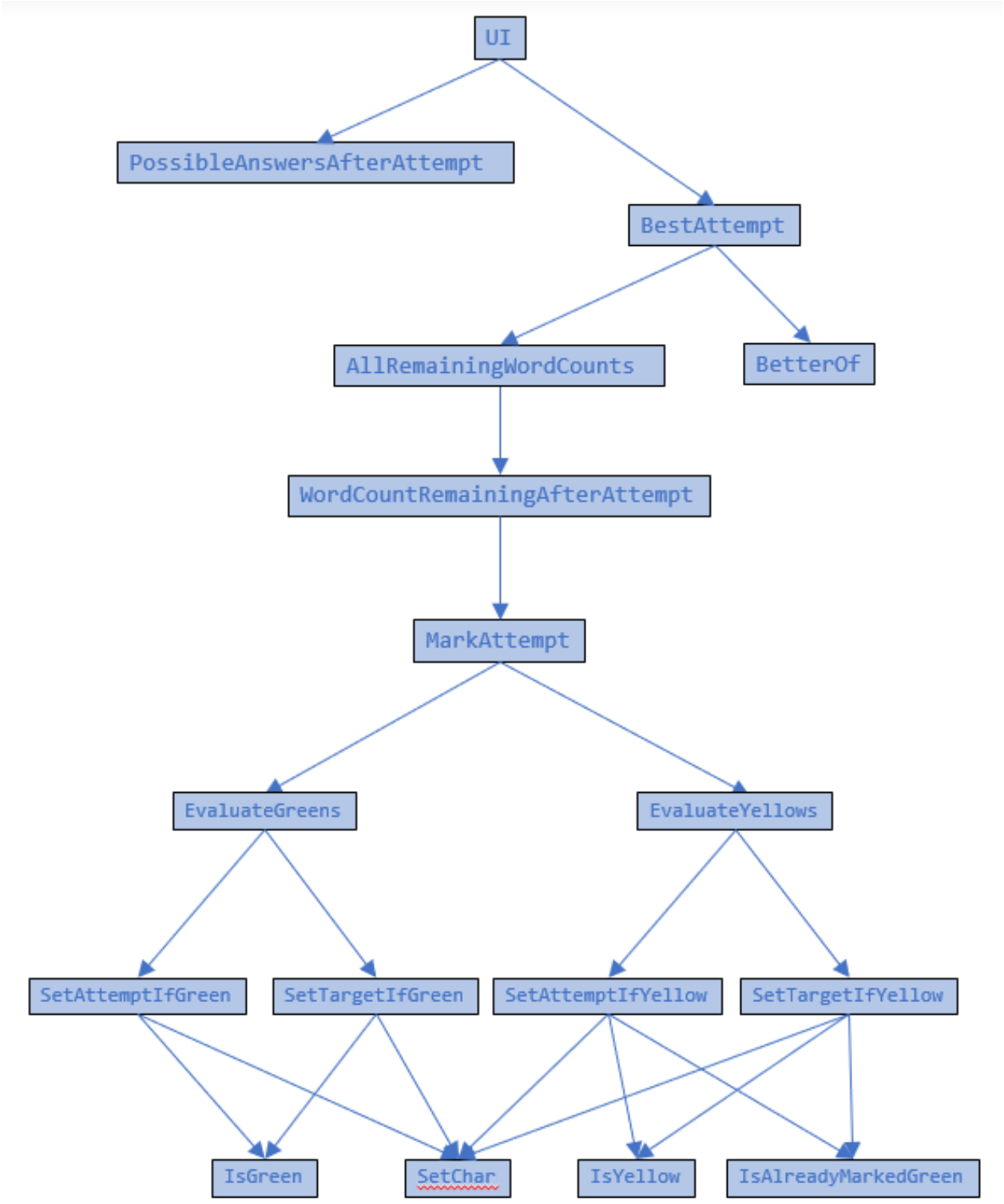
The core functionality will be called by an interactive program, designed to apply the Wordle Solver to the solution of a single Wordle puzzle – in particular to the daily online Wordle puzzle. The program will have a very simple, 'console', user interface that follows the sketch shown in the Analysis section (see Mock-up of the user interface). The required flow of the program is shown diagrammatically below:



For the last step in the loop, we will need a function named `PossibleAnswersAfterAttempt`, specified as follows:

Function Name	Arguments supplied	Returns
`PossibleAnswersAfterAttempt`	List of possible answer words prior to this next attempt. Attempt word Mark awarded (also known as 'outcome')	List of possible answers remaining from the input list that are still compatible with the mark awarded for the specified attempt word.

This 'main' program will call the core functions specified earlier. This design will ensure that the UI code calls the core functions, but never vice versa. This last point is essential to preserving the purity of the core functions. Also, this arrangement adopts the recognised design principle of 'separation of concerns'.[6] It means that the core functionality could easily be re-used with a different user interface, or called directly by a different program through an 'Application Programming Interface' (API). The complete 'dependency graph' is shown below:



[6] https://en.wikipedia.org/wiki/Separation_of_concerns

Technical Solution

Video showing the working solution

A video showing the working solution and with a brief overview of the code may be viewed here: https://www.loom.com/share/75b11691603d485fb7b2a356b4e51e94

Source code repository

Anyone wishing to run this program for themselves may obtain my complete solution from GitHub[7], including the full data definitions, and instructions on how to run the solution.

The final version of the resulting solution consists of just sixteen functions, each written as a single line of code (though several wrap to two or three lines on the page), plus just nine lines of procedural code defining the whole of the 'main program' - the Console user interface.

The complete code fits, comfortably, on just one page and is shown in full overleaf, with the exception of the constant data definitions (the two very long lists of `AllPossibleAnswers`and `ValidWords`) which are shown overleaf only as stubs.

The code is followed by a commentary, explaining the role of each function and the specific coding techniques employed.

[7] https://github.com/MetalUp/WordleSolver

Complete code for the Wordle Solver

```
using static System.Linq.Enumerable;

static bool IsGreen(string attempt, string target, int n) => target[n] == attempt[n];

static string SetChar(string word, int n, char newChar) =>
    word.Substring(0, n) + newChar + word.Substring(n + 1);

static string SetAttemptIfGreen(string attempt, string target, int n) =>
     IsGreen(attempt, target, n) ? SetChar(attempt, n, '*') : attempt;

static string SetTargetIfGreen(string attempt, string target, int n) =>
    IsGreen(attempt, target, n) ? SetChar(target, n, '.') : target;

static (string attempt, string target) EvaluateGreens(string attempt, string target) =>
    Range(0, 5).Aggregate((attempt, target), (a, n) =>
        (SetAttemptIfGreen(a.attempt, a.target, n), SetTargetIfGreen(a.attempt, a.target, n)));

static bool IsYellow(string attempt, string target, int n) => target.Contains(attempt[n]);

static bool IsAlreadyMarkedGreen(string attempt, int n) => attempt[n] == '*';

static string SetAttemptIfYellow(string attempt, string target, int n) =>
   IsAlreadyMarkedGreen(attempt, n) ? attempt : IsYellow(attempt, target, n) ?
       SetChar(attempt, n, '+') : SetChar(attempt, n, '_');

static string SetTargetIfYellow(string attempt, string target, int n) =>
   IsAlreadyMarkedGreen(attempt, n) ? target : IsYellow(attempt, target, n) ?
        SetChar(target, target.IndexOf(attempt[n]), '.') : target;

public static (string attempt, string target) EvaluateYellows(string attempt, string target) =>
    Range(0, 5).Aggregate((attempt, target), (a, n) =>
        (SetAttemptIfYellow(a.attempt, a.target, n), SetTargetIfYellow(a.attempt, a.target, n)));

public static string MarkAttempt(string attempt, string target) =>
    EvaluateYellows(EvaluateGreens(attempt, target).attempt,
        EvaluateGreens(attempt, target).target).attempt;

static IEnumerable<string> PossibleAnswersAfterAttempt(IEnumerable<string> prior, string attempt,
string mark) =>
    prior.Where(w => MarkAttempt(attempt, w) == mark).ToList();

static int WordCountRemainingAfterAttempt(IEnumerable<string> possibleAnswers, string attempt) =>
    possibleAnswers.GroupBy(w => MarkAttempt(attempt, w)).Max(g => g.Count());

static IEnumerable<(string word, int count)> AllRemainingWordCounts(IEnumerable<string> possAnswers,
IEnumerable<string> possAttempts) =>
    possAttempts.AsParallel().Select(w => (w, WordCountRemainingAfterAttempt(possAnswers, w)));

static (string word, int count) BetterOf((string word, int count) word1, (string word, int count)
word2, IEnumerable<string> possAnswers) =>
    (word2.count < word1.count) || (word2.count == word1.count && possAnswers.Contains(word2.word)) ?
        word2 : word1;

static string BestAttempt(IEnumerable<string> possAnswers, IEnumerable<string> possAttempts) =>
    AllRemainingWordCounts(possAnswers, possAttempts).
        Aggregate((bestSoFar, next) => BetterOf(bestSoFar, next, possAnswers)).word;

//Data definitions (stubs only - See Appendix I)
var ValidWords = new List<string> {};
var AllPossibleAnswers = new List<string> {};

//UI code:
var possible = AllPossibleAnswers;
var outcome = "";
while (outcome != "*****")
{
    var attempt = BestAttempt(possible, ValidWords);
    Console.WriteLine(attempt);
    outcome = Console.ReadLine();
    possible = PossibleAnswersAfterAttempt(possible, attempt, outcome).ToList();
}
```

Commentary on the core functional code

```
using static System.Linq.Enumerable;
```

This just allows the `Range` function to be used without having to qualify it as `Enumerable.Range` each time.

```
static bool IsGreen(string attempt, string target, int n) => target[n] == attempt[n];
```

Note that *all* the functions in the solution are marked `static` because they are standalone functions, not object instance methods.

```
static string SetChar(string word, int n, char newChar) =>
    word.Substring(0, n) + newChar + word.Substring(n + 1);
```

This is a simple 'helper' method. It is an example of using the DRY (Don't Repeat Yourself) principle: to avoid repeating the same chunk of code in four different later function calls.

```
static string SetAttemptIfGreen(string attempt, string target, int n) =>
    IsGreen(attempt, target, n) ? SetChar(attempt, n, '*') : attempt;
```

This uses the C# 'ternary operator', which may be read as:

'If `IsGreen(attempt, target, n) is true,` *return the result from calling* `SetChar(attempt, n, '*')`*, otherwise return the (unmodified)* `attempt`'.

```
public static string SetTargetIfGreen(string attempt, string target, int n) =>
    IsGreen(attempt, target, n) ? SetChar(target, n, '.') : target;
```

This is similar to `SetAttemptIfGreen` but applies to the `target` word, and replacing the match with the `.` character.

```
static (string attempt, string target) EvaluateGreens(string attempt, string target) =>
    Range(0, 5).Aggregate((attempt, target), (a, n) =>
        (SetAttemptIfGreen(a.attempt, a.target, n), SetTargetIfGreen(a.attempt, a.target, n)));
```

Here the 'marked up' versions of the input `attempt` and `target` words are returned as a *named* 2-tuple: `(string attempt, string target)`. The advantage of using a named tuple is that code that uses it may access individual elements by name – `attempt` or `target` – rather than by the generic properties names of `Item1`, `Item2`.

The implementation uses a higher-order function, `Aggregate`, which is the C# implementation of the generic 'fold' pattern. `Aggregate` is an extension method operating on any enumerable data structure (such as a list). Here, we apply aggregate to an enumerable of integer values `0` to `4` using the function `Range(0, 5)`.

The first argument passed into `Aggregate` is the starting point, also known as the 'seed' for the folding. Here it is a 2-tuple – `(attempt, target)` – holding the initial values of the two words. This is followed by a 'lambda':

```
(a, n) => (SetAttemptIfGreen(a.Item1, a.Item2, n), SetTargetIfGreen(a.Item1, a.Item2, n))
```

This lambda is applied to each member of the enumerable in turn. `a` represents the value of the 'aggregator' at each call, and `n` will be the next value from the enumerable (i.e. one of `0,1,2,3,4`). The lambda returns a 2-tuple made up of the marked up version of the attempt and target respectively – and this will replace the current value of the aggregator – `a`. We can see, then, that the aggregator always processes a tuple of two strings, starting with the 'seed' value, and ending as the attempt and target words with each green match marked as ***** and **.** respectively.

```
static bool IsYellow(string attempt, string target, int n) => target.Contains(attempt[n]);
```

`IsYellow` is equivalent to `IsGreen` but note that the implementation uses `Contains` to see if the specified attempt letter exists in the target word. This will only work correctly because any letter in the target word that has already been used by a green match, or a previous yellow match, will no longer be a letter: it will have been replaced by `.`

```
static bool IsAlreadyMarkedGreen(string attempt, int n) => attempt[n] == '*';
```

This is a simple 'helper' method. It is used only once, and hence saves no duplication, but it does make the next (more complex) function easier to read:

```
static string SetAttemptIfYellow(string attempt, string target, int n) =>
   IsAlreadyMarkedGreen(attempt, n) ? attempt : IsYellow(attempt, target, n) ?
       SetChar(attempt, n, '+') : Set(attempt, n, '_');
```

Broadly equivalent in *role* to `SetAttemptIfGreen`, the implementation of `SetAttemptIfYellow` uses two ternary operators, nested. This may be re-written as pseudo code:

> *If character number `n` in the attempt word is already marked green just return the `attempt` unmodified (as it can't also be yellow).*
>
> *If not, then if character `n` is yellow, set the character to `'+'` (indicating a yellow), otherwise set it to `'_'` (indicating a grey) because evaluation of that character in the attempt word has been completed with no matches.*

```
static string SetTargetIfYellow(string attempt, string target, int n) =>
   IsAlreadyMarkedGreen(attempt, n) ? target : IsYellow(attempt, target, n) ?
       SetChar(target, target.IndexOf(attempt[n]), '.') : target;
```

Similar to `SetAttemptIfYellow` but applying to the `target` word, and replacing the matched character with the `.` character.

```
public static (string attempt, string target) EvaluateYellows(string attempt, string target) =>
    Range(0, 5).Aggregate((attempt, target), (a, n) =>
        (SetAttemptIfYellow(a.attempt, a.target, n), SetTargetIfYellow(a.attempt, a.target, n)));
```

Equivalent to `EvaluateGreens`.

```
static string MarkAttempt(string attempt, string target) =>
    EvaluateYellows( EvaluateGreens(attempt, target).attempt,
        EvaluateGreens(attempt, target).target).target;
```

`MarkAttempt` calls `EvaluateYellows` on the results of calling `EvaluateGreens`. Note however that since the latter returns a 2-tuple, `EvaluateGreens` is being called twice and the two values in the tuple (being the modified `attempt` and `target` words respectively) are being extracted with `.attempt` and `.target`. Having to call the function twice is not elegant – it is the first example encountered in this project of current limitations in C# for supporting FP idioms. Pure FP languages such as Haskell typically have better patterns for deconstructing a tuple.

```
static IEnumerable<string> PossibleAnswersAfterAttempt(
    IEnumerable <string> prior, string attempt, string mark) =>
        prior.Where(w => MarkAttempt(attempt, w) == mark);
```

`PossibleAnswersAfterAttempt` is a surprisingly simple implementation, using the `.Where` function (the C# implementation of the generic 'filter' pattern). `Where` is applied to a the possible answers `prior` to the attempt, and applies a filter defined by the lambda `w => MarkAttempt(attempt, w) == mark` such that the list is filtered down to just those words that would have been marked the same was as the actual `mark` given to the attempt.

```
static int WordCountRemainingAfterAttempt(IEnumerable<string> possibleAnswers, string attempt) =>
    possibleAnswers.GroupBy(w => MarkAttempt(attempt, w)).Max(g => g.Count());
```

The role of this function is to count how many possible answer words would be left by the worst possible outcome from a given `attempt`, against a known set of `possibleWords`. (This is the *first* of the two algorithm variants proposed earlier – the other variant is shown in the Testing section - see Comparing the two variants of the algorithm, plus 'hard' mode). It does this by applying the `MarkAttempt` function to each of those possible words using the lambda `w => MarkAttempt(attempt, w))`. However, instead of applying this using the `Select` function (the C# implementation of the generic 'map' function) it uses `GroupBy`. The latter arranges all the input words into groups, where each word in a group would have the same result from applying `MarkAttempt`. The function `Max` is then applied to each of the *groups*, to identify the group with the maximum number of member words, using `g.Count()`.

```
static IEnumerable<(string word, int count)> AllRemainingWordCounts(IEnumerable<string> possAnswers,
IEnumerable<string> possAttempts) =>
    possAttempts.AsParallel().Select(w => (w, WordCountRemainingAfterAttempt(possAnswers, w)));
```

`IEnumerable<T>` is an interface that is implemented by any data structure that a calling function can iterate through by repeated asking for the 'next' element – `List` being one example. In this case the type `T` is a named 2-tuple – `(string word, int count)` . The `AllRemainingWordCounts` function uses `WordCountRemainingAfterAttempt` and `Select`, which is the C# implementation of the more general idea of 'map'. It uses these to generate an enumerable where the type is a named 2-tuple `(string word, int count)` so that each possible attempt word is paired with the number of possible answers that would remain from the worst outcome if that attempt word was used.

```
static (string word, int count) BetterOf((string word, int count) word1, (string word, int count)
word2, IEnumerable<string> possAnswers) =>
    (word2.count < word1.count) || (word2.count == word1.count && possAnswers.Contains(word2.word)) ?
        word2 : word1;
```

This is a simple function that chooses the better of two words being considered as the next attempt. In general it will choose the word which has the lower `count` i.e. that would leave the least remaining possible answers from the worst outcome (mark). However, should the two words both have the *same* count then it will favour the one (if any) that is *also* a possible answer word.

```
static string BestAttempt(IEnumerable<string> possAnswers, IEnumerable<string> possAttempts) =>
    AllRemainingWordCounts(possAnswers, possAttempts).Aggregate((bestSoFar, next) =>
        BetterOf(bestSoFar, next, possAnswers)).word;
```

The final, top-level, function now takes the results from `AllRemainingWordCounts` and finds the best one. It uses `Aggregate` ('fold/reduce') to go through all the possible attempts applying the `BetterOf` function to compare the `next` to the `bestSoFar` (both being 2-tuples). Finally it extracts the `word` part of the final `bestSoFar` 2-tuple to return the best attempt as a string.

Commentary on the user interface code

We can now look at how these functions are used within the user interface (procedural code)

The code first defines and initializes two ‘variables’:

```
var possible = AllPossibleAnswers;
var outcome = "";
```

These are conventional ‘procedural style’ variables meaning that the same variable may be re-assigned with different values during the run. (In FP, the term ‘variables’ refers to parameters – the value of which will typically vary between different calls to the function, but in each such call it is actually a different variable. FP ‘variables’ are never re-assigned). We need these procedural-style variable in the UI in order to run the procedural loop that follows with the same variables having different values each pass of the loop.

```
while (outcome != "*****")
{
   …
}
```

This loop simply executes *until* an outcome of `"*****"` (five greens) has been achieved. It deliberately does not enforce the Wordle rule of maximum six attempts because I want to know how many attempts the current algorithm would taken even if it failed to identify the target within six attempts.

```
var attempt = BestAttempt(possible, ValidWords);
```

First time around the loop the first argument will be list of all 2309 possible answers but with each subsequent iteration this list will (hopefully) have been filtered down further. However, the second argument will always be the fixed list of 12,947 **valid words**. This makes for an expensive calculation but it means that the algorithm always has the option to use any word that would most reduce the remaining words, irrespective of whether that word was a possible answer itself.

Incidentally, because all the functions are deterministic, the calculation of the best ‘first attempt’ should produce the same word each time the program is run: so, like many human players, the program will have a ‘favourite’ start word. It will be interesting to see which word it chooses.

```
    Console.WriteLine(attempt);
    outcome = Console.ReadLine();
```

Write the attempt word and then read in the user-entered outcome (marks).

```
possible = PossibleAnswersAfterAttempt(possible, attempt, outcome).ToList();
```

Recalculate the possible answers based on the `attempt` and the user specified `outcome`.

Testing

The Wordle solver program was extensively tested, and at five different levels.

1) Every single function was unit tested, as it was written.
2) The whole program was tested to ensure that it ran correctly and that it was actually capable of solving the official, daily Wordle puzzle – 10 days in a row.
3) The program was tested for performance – measuring how long the computations took – and opportunities for improving the performance were explored.
4) With the benefit of the performance optimisations above, the effectiveness of the Solver was tested by building a small test loop that would simulate the solving of a Wordle puzzle *for each one of the possible 2309 target words* that may be set. This would allow the compilation of accurate statistics, showing the percentage of target words that the program could solve within six guesses, the breakdown within that of 1, 2, 3, 4, 5, or 6 attempts, and the average number of attempts.
5) The two variants of the algorithm were implemented and their effectiveness compared, then the better of the two was also run against a simulation of Wordle in 'Hard mode'.

The following sub-sections expand on each of these five levels of testing.

Writing executable unit tests for all functions

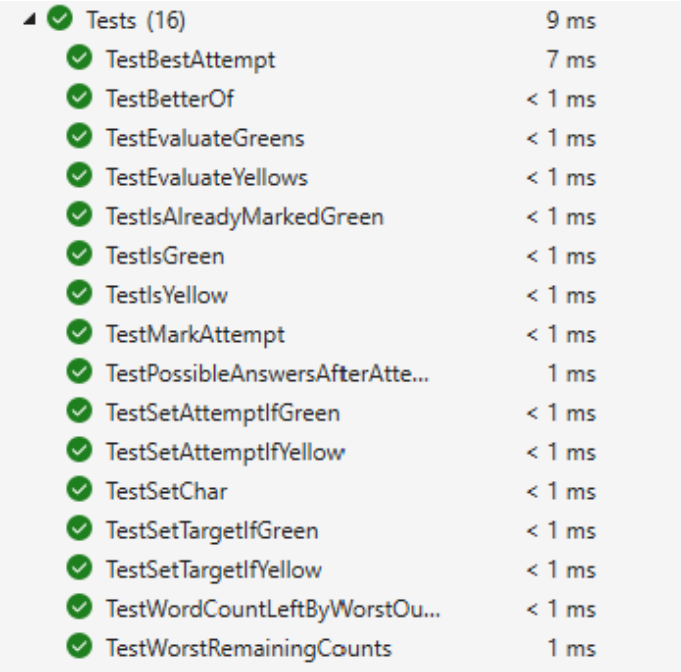
All the functions have been thoroughly unit tested, by writing unit tests that execute under an automated unit testing framework (NUnit). This is good practice for any software development, not only to prove that the code works correctly but to permit functions to be 're-factored' (improved in the implementation without changing the functionality) without fear of breaking the system. However, unit testing is especially relevant to functional programming for three reasons:

1) It is easier to write unit tests for pure functions, because such functions have no external dependencies, nor any side effects: each test just needs to call the function with specified parameters and then compare the returned result to the value that would be expected.

2) There is a higher reward from unit testing in FP, because if pure functions **A** and **B** are both correct, then the 'combination' of those functions is correct. This is not true of less rigorous forms of coding – because of potential side effects and/or external dependencies. One of the several problems resulting from mixing up core functionality with input/output code is that it becomes *extremely* difficult to write executable unit tests.

3) One downside to FP is that debugging of an individual functions can be harder. With *procedural* coding it is easy to put a breakpoint within a loop, in FP it is not always possible to break into an iteration implemented using a `map`, `filter`, `fold` or other higher level function operating on an enumerable. That said, using those higher order functions means that you write much less code, and have much less scope for making the same kind of errors (e.g. 'off-by-one' errors) that are commonly made in procedural coding. Thorough unit testing gives you the confidence *that* your function is working correctly from the outside-in rather than working from the inside-out.

Each test covers one function, but with multiple test cases. The screenshot below shows all the tests passing ('in the green') running in Visual Studio. This is followed by the code for all 16 test methods.



```
[TestMethod]
public void TestIsGreen()
{
    Test("ABCDE", "A____", 0, true);
    Test("ABCDE", "____E", 4, true);
    Test("ABCDE", "_A___", 1, false);
    Test("BABBB", "B____", 1, false);

    void Test(string attempt, string target, int charNo, bool expected)
    {
        Assert.AreEqual(expected, IsGreen(attempt, target, charNo));
    }
}

[TestMethod]
public void TestSetChar()
{
    Test("ABCDE", 0, '_', "_BCDE");
    Test("ABCDE", 4, '_', "ABCD_");

    void Test(string word, int charNo, char newChar, string expected)
    {
        Assert.AreEqual(expected, SetChar(word, charNo, newChar));
    }
}

[TestMethod]
public void TestSetAttemptIfGreen()
{
    Test("ABCDE", "ABCDE", 0, "*BCDE");
    Test("ABCDE", "ABCDE", 4, "ABCD*");
    Test("BBCDE", "ABCDE", 0, "BBCDE");
    Test("ABCDE", "AACDE", 0, "*BCDE");

    void Test(string attempt, string target, int charNo, string expected)
    {
        Assert.AreEqual(expected, SetAttemptIfGreen(attempt, target, charNo));
    }
}

[TestMethod]
public void TestSetTargetIfGreen()
{
    Test("ABCDE", "ABCDE", 0, ".BCDE");
    Test("ABCDE", "ABCDE", 4, "ABCD.");
    Test("BBCDE", "ABCDE", 0, "ABCDE");
    Test("ABCDE", "AACDE", 0, ".ACDE");

    void Test(string attempt, string target, int charNo, string expected)
    {
        Assert.AreEqual(expected, SetTargetIfGreen(attempt, target, charNo));
    }
}

[TestMethod]
public void TestEvaluateGreens()
{
    Test("ABCDE", "AXXXX", ("*BCDE", ".XXXX"));
    Test("ABCDE", "XXXXE", ("ABCD*", "XXXX."));
    Test("ABCDE", "ABCDE", ("*****", "....."));
    Test("AACDE", "AXXXX", ("*ACDE", ".XXXX"));
    Test("ABCDE", "AAXXX", ("*BCDE", ".AXXX"));

    void Test(string attempt, string target, (string, string) expected)
    {
        Assert.AreEqual(expected, EvaluateGreens(attempt, target));
    }
}
```

```
[TestMethod]
public void TestIsYellow()
{
    Test("ABCDE", "____A", 0, true);
    Test("ABCDE", "____A", 4, false);
    Test("ABCDE", "___AA", 0, true);
    Test("AACDE", "_A___", 1, true);
    Test("AACDE", "__A__", 1, true);

    void Test(string attempt, string target, int charNo, bool expected)
    {
        Assert.AreEqual(expected, IsYellow(attempt, target, charNo));
    }
}

[TestMethod]
public void TestIsAlreadyMarkedGreen()
{
    Test("AB*DE", 2, true);
    Test("AB*DE", 0, false);
    Test("AB*DE", 4, false);
    Test("*BCD*", 2, false);
    Test("*BCD*", 0, true);
    Test("*BCD*", 4, true);

    void Test(string attempt, int n, bool expected)
    {
        Assert.AreEqual(expected, IsAlreadyMarkedGreen(attempt, n));
    }
}

[TestMethod]
public void TestSetAttemptIfYellow()
{
    Test("ABCDE", "EABCD", 0, "+BCDE");
    Test("ABCDE", "EABCD", 4, "ABCD+");
    Test("ABCDE", "BAAAA", 0, "+BCDE");
    Test("AAAAB", "EABBB", 4, "AAAA+");

    void Test(string attempt, string target, int charNo, string expected)
    {
        Assert.AreEqual(expected, SetAttemptIfYellow(attempt, target, charNo));
    }
}

[TestMethod]
public void TestSetTargetIfYellow()
{
    Test("ABCDE", "EABCD", 0, "E.BCD");
    Test("ABCDE", "EABCD", 4, ".ABCD");
    Test("ABCDE", "BAAAA", 0, "B.AAA");
    Test("AAAAB", "EABEA", 4, "EA.EA");
    Test("AAAAB", "EABBB", 4, "EA.BB");
    Test("*BCDE", "*BCDA", 4, "*BCDA");

    void Test(string attempt, string target, int charNo, string expected)
    {
        Assert.AreEqual(expected, SetTargetIfYellow(attempt, target, charNo));
    }
}
```

```
[TestMethod]
public void TestEvaluateYellows()
{
    Test("ABCDE", "XAXXX", ("+____", "X.XXX"));
    Test("ABCDE", "XXXXA", ("+____", "XXXX."));
    Test("ABCDE", "XXXXE", ("____+", "XXXX."));
    Test("ABCDE", "XAAXX", ("+____", "X.AXX"));
    Test("AACDE", "XAXXX", ("+____", "X.XXX"));
    Test("ABCDE", "BCDEA", ("+++++", "....."));

    void Test(string attempt, string target, (string, string) expected)
    {
        Assert.AreEqual(expected, EvaluateYellows(attempt, target));
    }
}

[TestMethod]
public void TestMarkAttempt()
{
    Test("ABCDE", "XXXXX", "_____");
    Test("ABCDE", "BCDEA", "+++++");
    Test("ABCDE", "ABCDE", "*****");
    Test("SAINT", "LADLE", "_*___");
    Test("IDEAL", "LADLE", "_++++");
    Test("CABAL", "RECAP", "+__*_");
    Test("CABAL", "RECAP", "+__*_");
    Test("COLON", "GLORY", "_++__");

    void Test(string attempt, string target, string expected)
    {
        Assert.AreEqual(expected, MarkAttempt(attempt, target));
    }
}

[TestMethod]
public void TestPossibleAnswersAfterAttempt()
{
    var prior = new List<string> { "ABCDE", "BCDEA", "CDEAB", "DEABC", "EABCD" };
    Test(prior, "AAAAA", "*____", "ABCDE");
    Test(prior, "AXXXX", "+____", "BCDEA", "CDEAB", "DEABC", "EABCD");
    Test(prior, "AXXBX", "+__+_", "BCDEA", "CDEAB", "EABCD");

    void Test(List<string> prior, string attempt, string mark, params string[] expected)
    {
        CollectionAssert.AreEqual(expected, PossibleAnswersAfterAttempt(prior, attempt,
mark).ToList());
    }
}

[TestMethod]
public void TestWordCountRemainingAfterAttempt()
{
    var prior = new List<string> { "ABCDE", "BCDEA", "CDEAB", "DEABC", "EABCD" };
    Test(prior, "AAAAA", 1);
    Test(prior, "AXXXX", 4);
    Test(prior, "XXXXX", 5);

    void Test(List<string> prior, string attempt, int expected)
    {
        Assert.AreEqual(expected, WordCountRemainingAfterAttempt(prior, attempt));
    }
}
```

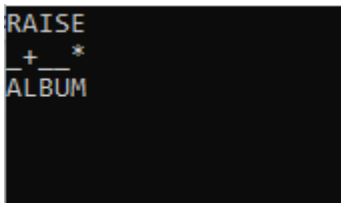
```
[TestMethod]
public void TestAllRemainingWordCounts()
{
    var possAnswers = new List<string> { "AAAAA", "BBBBB", "CCCCC", "DDDDD" };
    var possAttempts = new List<string> { "ABABA", "BCBCB", "ABCBC" };
    var expected = new List<(string word, int count)> { ("ABABA", 2), ("BCBCB", 2), ("ABCBC", 1) };
    Test(possAnswers, possAttempts, expected);

    void Test(List<string> possAnswers, List<string> possAttempts, List<(string word, int count)>
expected)
    {
        CollectionAssert.AreEqual(expected, AllRemainingWordCounts(possAnswers,
possAttempts).ToList());
    }
}

[TestMethod]
public void TestBetterOf()
{
    var possAnswers = new List<string> { };
    Test(("A", 3), ("B", 2), possAnswers, "B");
    Test(("B", 2), ("A", 3), possAnswers, "B");
    Test(("B", 2), ("A", 2), possAnswers, "B");
    Test(("A", 2), ("B", 2), possAnswers, "A");
    possAnswers = new List<string> { "B" };
    Test(("A", 2), ("B", 2), possAnswers, "B");
    possAnswers = new List<string> { "B", "A" };
    Test(("A", 2), ("B", 2), possAnswers, "B");
    Test(("B", 2), ("A", 2), possAnswers, "A");

    void Test((string word, int count) word1, (string word, int count) word2, List<string>
possAnswers, string expected)
    {
        Assert.AreEqual(expected, BetterOf(word1, word2, possAnswers).word);
    }
}

[TestMethod]
public void TestBestAttempt()
{
    var possAnswers = new List<string> { "ABCDE", "ABBBB", "EDCBA" };
    var possAttempts = new List<string> { "AAAAA", "BBBBB", "CCCCC", "DDDDD", "EEEEE", "EDCBA",
"DEABC" };
    Test(possAnswers, possAttempts, "EDCBA");
    possAnswers = new List<string> { "ABCDE", "ABBBB", "BCDEA" };
    possAttempts = new List<string> { "AAAAA", "BBBBB", "CCCCC", "DDDDD", "EEEEE", "EDCBA", "DEABC" };
    Test(possAnswers, possAttempts, "BBBBB");

    void Test(List<string> possAnswers, List<string> possAttempts, string expected)
    {
        Assert.AreEqual(expected, BestAttempt(possAnswers, possAttempts));
    }
}
```

Testing that the program can solve the daily Wordle puzzle

The first time I ran the program, it popped up an empty Console window with a flashing cursor and nothing else happened! An agonizing thirty seconds later, the screen changed to:

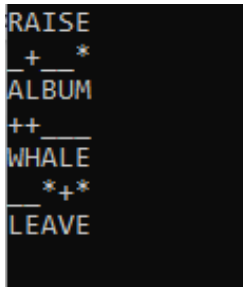
The Wordle Solver had found its best start word: **RAISE**. I opened the daily online Wordle game (**No. 469 – 1 Oct 2022**), and typed in **RAISE**. Wordle responded with:

Back on the Wordle Solver console I translated the colours to the required symbols **_ + _ _ *,** typed them in and hit Enter. It came back with the second attempt word **ALBUM** much more quickly:

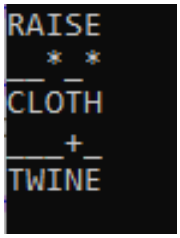


Interesting choice! I knew that the algorithm would not necessarily pick a word that was compatible with the outcomes so far – and this was confirmation.

After just two more attempts and Wordle Solver had got to the answer.



The next morning I tried it again (**No. 470 – 2 Oct 2022**). Here's the result:



This seemed almost too good to be true – after the second guess it still only had the letters **T**, **I**, and **E**, yet it seemed to guess **TWINE** correctly. Had it just happened to pick out the right word from many possibilities? I decided to tweak the console program, adding one line at the end of the main loop:

```
Console.WriteLine($"{possible.Count} remaining");
```

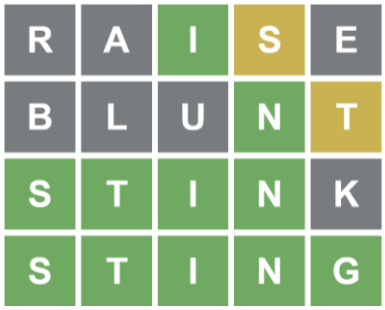
This would now tell me – after each attempt had been marked – how many target words remained as *possible* answers. I re-ran the first two games:

```
RAISE
_+__*
41 remaining
ALBUM
++___
2 remaining
WHALE
__*+*
1 remaining
LEAVE
```

```
RAISE
__*_*
23 remaining
CLOTH
___+_
1 remaining
TWINE
```

This shows that, at least for these two games, the algorithm had narrowed down the possibilities to just one, in both cases. This would be true for six of the next eight games also.

Wordle 471 - 3 Oct 2022




```
RAISE
__*+_
29 remaining
BLUNT
___*+
2 remaining
STINK
****_
1 remaining
STING
```

Wordle 472 – 4 Oct 2022

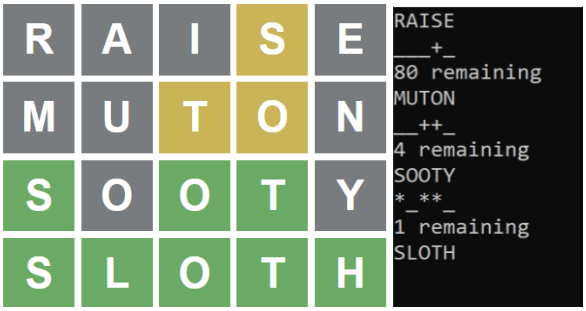

```
RAISE
_____
167 remaining
BLUDY
*_*__
1 remaining
BOUGH
```

Wordle 473– 5 Oct 2022


```
RAISE
+*_*_
2 remaining
MARSH
```

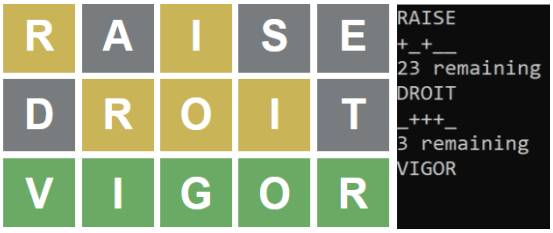
With just HARSH and MARSH remaining the solver could not narrow the range any further and just picked one of the two options.

Wordle 474 – 6 Oct 2022



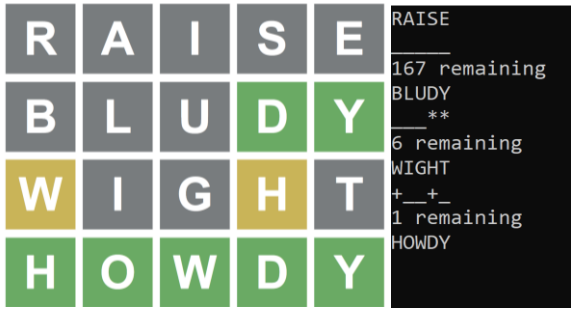
Wordle 475 – 7 Oct 2022

Wordle 476 – 8 Oct 2022



The 3 remaining here were MICRO, MINOR, and VIGOR. Choosing any of the three would not only have a 1/3 chance of being right, but if it was wrong the remaining two would have been reduced to just one.

Wordle 477 – 9 Oct 2022



Wordle 478 – 10 Oct 2022

In the last example, there were just 3 remaining after the second attempt: **EBONY**, **ENVOY**, and **ENJOY**, each with a 1/3 chance of being right. However, by selecting **ENVOY** the remainder was reduced to 1. (**ENJOY** would also have worked, but **EBONY** as the third guess would – if it had been wrong – have left 2 remaining possibilities.)

Overall, the results from these first 10 tests against the live online Wordle game were pleasing: 1 x 2 attempts, 3 x 3 attempts, 5 x 4 attempts, and 1 x 5 attempts. However, this is far too small a sample from which to extrapolate the solver's 'form'. The next stage would be to run a sample over hundreds of games, ideally over all 2309 possible target words.

However, I decided to look at performance next, because processing time would become more critical in a simulation of many games.

(The video, mentioned earlier, shows Wordle Solver tackling the online Wordle puzzle for 03/12/2022. View it here: https://www.loom.com/share/75b11691603d485fb7b2a356b4e51e94)

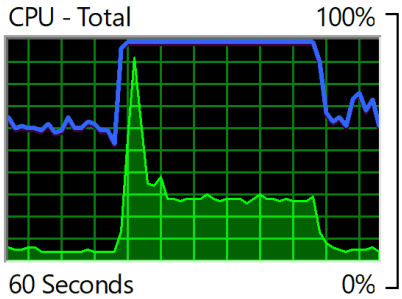
Improving the performance through parallel processing

The following code snippet measures the time to determine the first attempt accurately:

```
var sw = new Stopwatch();
sw.Start();
BestAttempt(AllPossibleAnswers, ValidWords);
sw.Stop();
Console.WriteLine($"{sw.ElapsedMilliseconds} ms");
```

It yielded the result of **32,998 ms**.

I ran the same code again again using the **Windows Resource Monitor** to show processor usage:

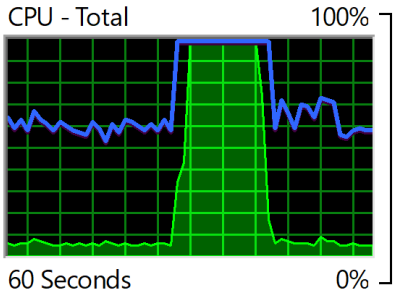


This showed that, after the inevitable spike when a new program is started up, for the remainder of the run overall processor usage was between 25 and 30%. This makes sense: my laptop has four cores and the main program is effectively running on just one core (the bit over 25% can be accounted for by background system processes).

Next I made the a tiny change to the `AllRemainingWordCounts` function, inserting a call to `AsParallel`:

```
public static IEnumerable<(string word, int count)> AllRemainingWordCounts(IEnumerable<string>
possAnswers, IEnumerable<string> possAttempts) =>
        possAttempts.AsParallel().Select(w => (w, WordCountRemainingAfterAttempt(possAnswers, w)));
```

This time processor usage shot up to 100% and remained there until the function had completed:



The **elapsed time fell to 11,527 ms** – down by almost a factor of three. My understanding from reading up on 'Parallel LINQ' (known informally as 'plinq') is that you would never see a speed up factor of four (for a four core machine) – because of the background systems processes. A speed up of just under three seems slightly disappointing, though. In a small side-investigation, I wrote the following three functions to generate a list of prime numbers:

```
static List<int> PrimesUpTo(int n) => Enumerable.Range(2, n - 1).AsParallel().Where(x =>
IsPrime(x)).ToList();

static bool IsPrime(int n) => !Enumerable.Range(2, n / 2 - 1).Any(f => IsFactor(f, n));

static bool IsFactor(int f, int n) => n % f == 0;
```

Calling this function with an argument of 1,000,000, with and without the `AsParallel()`, I was able to show that the latter delivered a speed up of 3.8x on my four-core laptop, which was much nearer the (impossible) ideal of 4x.

Could the parallelization of Wordle Solver be further improved then? Several online postings suggested that it is all too easy to 'over parallelise' and this can have the opposite effect. I tried adding `AsParallel` into further functions that enumerated over large lists, but the result was that the program actually ran *slower* than with no parallelisation at all.

The posters had been right: optimization for parallel processing is still something of a 'dark art'. An interesting area to investigate further, but beyond the scope of this project.
I decided to take the near 3x improvement and leave it at that.

Before running a simulation involving many games there is one obvious piece of 'low-hanging fruit' to pick: given that the algorithm *always* chooses **RAISE** as its first attempt, this might as well be hard-coded (a rather extreme form of caching!). The console UI was changed, initializing the `attempt` variable up front and moving the evaluation of `BestAttempt` from the start to the end of the loop:

```
var possible = AllPossibleAnswers;
var outcome = "";
var attempt = "RAISE";
while (outcome != "*****")
{
    Console.WriteLine(attempt);
    outcome = Console.ReadLine();
    possible = PossibleAnswersAfterAttempt(possible, attempt, outcome).ToList();
    Console.WriteLine($"{possible.Count} remaining");
    attempt = BestAttempt(possible, ValidWords);
}
```

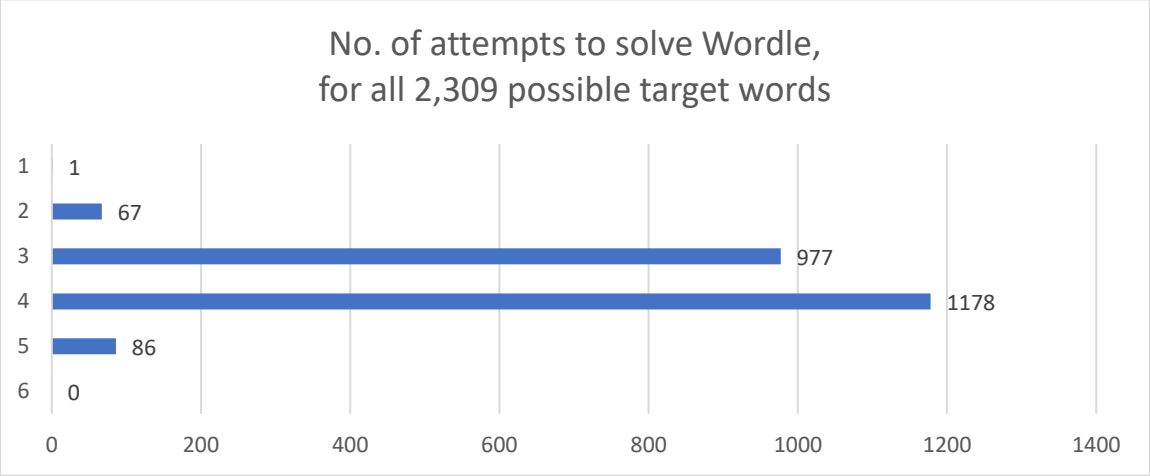
The UI appears unchanged, but the first attempt **RAISE** now appears almost instantaneously.

Testing the effectiveness of the algorithm, exhaustively

The parallelization combined with the hard-coded first attempt (though still using the attempt word calculated by the algorithm) meant that performance was now almost certainly fast enough to attempt an *exhaustive* evaluation: running the solver for 2,309 games covering each of the possible target words. The following small program was used to run the simulation. It loops through all 2,309 possible words, setting up each as the `target` in turn. It then runs a loop similar to the console UI program run previously, but instead of asking the user to enter the outcome, it just calls the `MarkAttempt` function. The loop also counts the number of attempts taken in the game. When this loop ends, the count is recorded into a dictionary, and the entries in this dictionary are displayed at the end:

```
var dict = new Dictionary<int, int>();
foreach(var target in AllPossibleAnswers)
{
    var possible = AllPossibleAnswers;
    var outcome = "";
    var attempt = "RAISE";
    int count = 0;
    while (outcome != "*****")
    {
        count++;
        outcome = MarkAttempt(attempt, target);
        possible = PossibleAnswersAfterAttempt(possible, attempt, outcome).ToList();
        attempt = BestAttempt(possible, ValidWords);
    }
    if (dict.Keys.Contains(count))
    {
        dict[count]++;
    } else
    {
        dict[count] = 1;
    }
}
foreach(var attempts in dict.Keys.OrderBy(k => k))
{
    Console.WriteLine($"{attempts} attempts: {dict[attempts]} games");
}
```

The simulation took 20 minutes to complete. The results are presented below in a bar chart – a form that will be familiar to any regular Wordle user:



The results were (at least to me) quite startling. Not only did the Wordle Solver *always* solve the puzzle within six attempts: itnever required more than *five* attempts. It achieved a 'hole in one' just

once, because **RAISE** is a possible answer. Processing these results further we get the following percentages:

Attempts	%	Cumulative
1	-	
2	2.9%	
3	42.3%	45.2%
4	51.0%	96.2%
5	3.7%	100%
6	-	

and an **average** of **3.55** attempts per puzzle.

Comparing the two variants of the algorithm, plus 'hard' mode

Having built the program to test the Solver exhaustively, the next task was to compare the performance of the two variants (of the same algorithm) proposed in the analysis section:

1. Return the number of remaining possible answers resulting from the *worst* outcome (mark) given to the attempt word.
2. Return the (mathematically) *expected* word count remaining after the attempt – effectively the weighted average of all possible outcomes.

and then, using whichever of the two variants proved more effective, I tested the Solver in simulated 'Hard mode'.

Variant 2 above, required only that the `WordCountRemainingAfterAttempt` function be changed to:

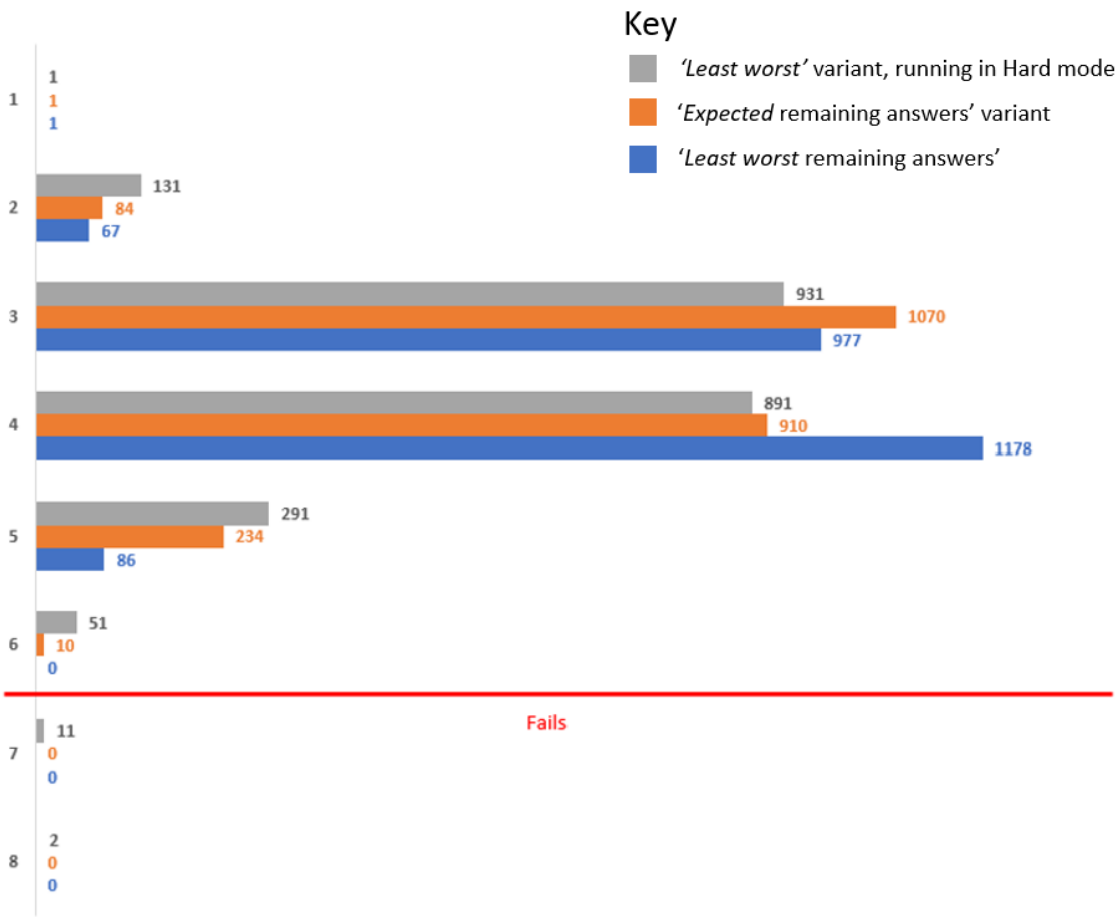
```
public static int WordCountRemainingAfterAttempt(IEnumerable<string> possibleAnswers, string attempt)
=>
    possibleAnswers.GroupBy(w => MarkAttempt(attempt, w)).Sum(g => g.Count() * g.Count());
```

For hard mode, it was necessary only to change the code for `BestAttempt`, changing the start of the expression from the (larger and fixed-size) `possAttempts` to the (smaller and reducing with each attempt) list of `possAnswers`:

```
public static string BestAttempt(IEnumerable<string> possAnswers, IEnumerable<string> possAttempts) =>
    AllRemainingWordCounts(possAnswers, possAnswers).
        Aggregate((bestSoFar, next) => BetterOf(bestSoFar, next, possAnswers)).word;
```

Unsurprisingly, running in Hard mode, was much faster, because the algorithm is evaluating far fewer possible attempt words each time.

The results for the three algorithms are summarised in the chart overleaf:



The second variant ('Expected ..') solved puzzle *more* often in 3 attempts than 4, but 10 of the possible target words took six attempts. The average number of attempts was 3.57 – *slightly* worse than the first variant ('Least worst')

Operating in Hard mode (using the better of the two variants: 'least worst') resulted in the puzzle being solved in just 2 attempts significantly more often than either of the other two. However, it failed to solve the puzzle in six attempts for 13 of the possible target words, and the average number of attempts was 3.67.

In summary, the first algorithm proposed for this project is the best performing in two measures:

- It is guaranteed to solve the puzzle in 5 attempts or fewer, and with 96% of games taking four or fewer attempts.
- Its overall average of 3.55 attempts per game is the lowest of the three tested algorithms

Evaluation

At the end of the Analysis section, 10 SMART objectives were set. These are repeated below (in abbreviated form), showing that 9 of the objectives were achieved, and the other one exceeded.

Evaluation against the SMART objectives

Objectives concerning functionality and usability of the resulting system

1. The system should be capable of solving the daily online Wordle puzzle, requiring the investigating user only to map the inputs and outputs between the two systems, not to provide any other assistance. **ACHIEVED**

2. Prove that, across all possible target words for Wordle, the system will solve the puzzle in six or fewer attempts in at least 97% of cases (gauged – from the interviews - to be equivalent to a reasonably good human player). **EXCEEDED** (it solves the puzzle in *five* of fewer attempts 100% of the time).

3. Demonstrate the difference between adopting the 'worst case' and 'expected' alternative variants of the algorithm (in terms of percentage of puzzles solved in six attempts, and the average number of attempts taken, across all 2,309 possible target words). **ACHIEVED**

4. Show that the algorithm could also solve puzzles in the 'Hard mode' setting, and determine the effectiveness relative to 2, using the same criteria as given in 3. **ACHIEVED**

5. Demonstrate that the minimalist user interface sketched previously, while designed only for the author's use, could be used by an external validator following simple written instructions. **ACHIEVED**

Objectives concerning the implementation

6. All core functionality (everything except the minimal user interface code) to be provided by 'pure' functions. **ACHIEVED**

7. All core functions to comprise a single statement returning the value of an expression. **ACHIEVED**

8. All core functions to be implemented without need for recursive calls. **ACHIEVED**

9. Demonstrate that the functional implementation can be parallelised for faster performance. **ACHIEVED**

10. Provide 100% unit testing coverage for core functions. **ACHIEVED**

External validation

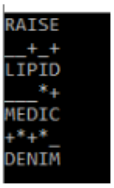
It was important to me to gain external validation of the Wordle Solver – not to establish how well it worked (I'd already done that) but to validate:

1) That the program works i.e. can solve a Wordle puzzle
2) That the program consists of nothing more than the functions shown in the
3) That there are no dependencies on any external frameworks
4) That there is no form of 'cheating' in the code.

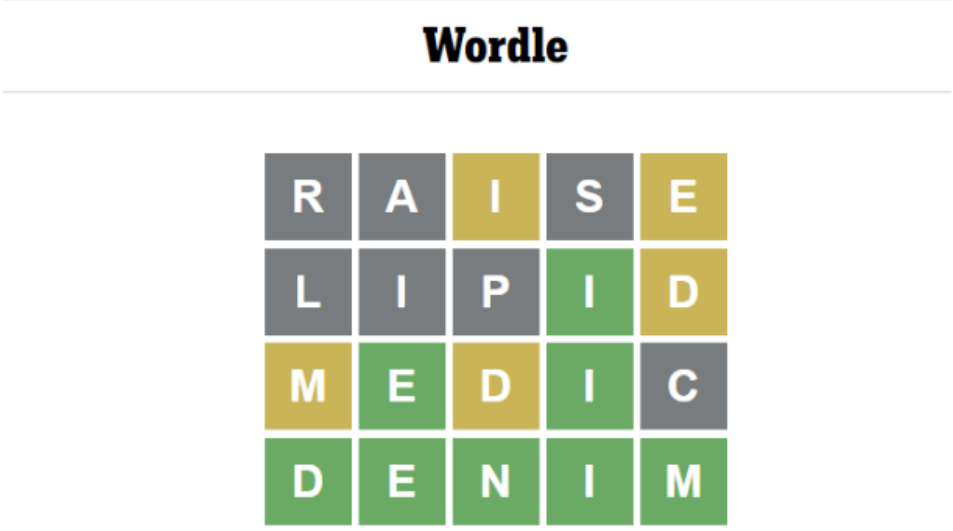
For this I approached Pete Dring, Head of Computing at Fulford School in York. I sent him the complete solution with minimal written instructions on how to run it.

In an email response on 20th October 2022, he replied that he had verified all these points, and he included the screenshots below in relation to the first:

Here's the console input/output:



and corresponding Wordle page:



Mr.Dring went on to say:

> *This is very impressive! … There's a huge amount of thought behind those … functions: it's a very elegant solution!*

Evaluating the user interface

Mr Dring, who did not see this report when he examined the solution, did comment on the limited user interface, though he had no difficulty using it. He also accepted my argument that the system was designed solely for my own use in conducting a scientific investigation into the feasibility of solving Wordle automatically, not for use by a gamer. As such, as I argued in Mock-up of the user interface, the current user interface is both effective and efficient.

Undoubtedly the user interface *could* be improved.

It could tell you, when the puzzle is complete (i.e. you have entered ********* as the mark), in case you didn't realise and carried on guessing **.**

There could be more validation. At present the UI does not check that the marks entered consist only of the three symbols *****, **+**, **_** (for Green, Yellow, Grey). Nor does it check for impossible combinations such as ***+*****. But if you enter wrong symbols or an invalid combination then the system isn't going to solve the puzzle. Either on that attempt or a subsequent one the solver will throw an exception – and not an especially 'friendly' one at that – when it determines that there are no possible remaining answers. But what have you lost? Just run the program again and enter the correct marks this time. Importantly, there is little point in adding these simple forms of validation when it is *impossible* to verify that the user has in fact entered the correct mark for the computer's attempt, because the whole point is that *the computer does not know the target word against which it could validate the marking*. If you enter a valid combination of the symbols, but not the correct ones, the computer will – either on the next attempt or shortly thereafter – reach a situation where there are no possible answers left that are consistent with all the marks.

It could be given a Graphical User Interface, perhaps echoing the design of the Wordle game itself. The technical architecture – which fully separates the concerns of the user interface from the core domain logic – would facilitate the implementation of a brand new user interface, including a GUI for use on a laptop or, like Wordle itself, an App for a mobile phone.

But what would be the point of designing a friendly user interface for this system? Could it be turned into a consumer product: an automated Wordle solver perhaps, or even, with the functionality deliberately reduced, a 'Wordle helper'? (The latter could, for example, show you the possible answer words remaining after your attempts so far). There's just no point: if you want *general* advice on how to improve your Wordle skills, there's plenty of such advice online (good and bad). If you just want to publish a great running average score, there are easier and even more effective ways to cheat – as described above.

In fact, if you want to cheat you can even guarantee to get the daily Wordle puzzle *in the first attempt every time*! An enterprising software engineer has reverse engineered the JavaScript code and published a spoiler list[8] - showing the daily answers for the next several years! -(Rather surprisingly, it turns out that the Wordle software runs entirely within the browser. A more intelligent design would have been to share the load between client and server, keeping the marking – and the knowledge of the target word – held securely on the server side.)

Having written the solver, I enjoy watching it perform against the Wordle puzzle. But I also still enjoy solving the puzzle for myself, without help to remember to do this before applying the Wordle

[8] https://medium.com/@owenyin/here-lies-wordle-2021-2027-full-answer-list-52017ee99e86

Solver. Solving it 'with just a little bit of help from the solver' doesn't deliver the satisfaction of either.

Evaluating the code style

I have made every effort to adopt best practices for code styling including:

- Use of long, descriptive names for functions and variables, especially where there are two or more parameters that might easily be confused. (Where I have used short names, for example using `n` for a count these have always been conscious choices).
- Minimising the duplication of code (the 'DRY' principle – as in Don't Repeat Yourself')
- Breaking out functions, even where there is only a single use, in the sole interest of making an otherwise complex function easier to read. (The principle of 'intentional coding')

One reviewer criticised the fact that the code uses literal characters instead of constants, for example:

```
static string SetAttemptIfGreen(string attempt, string target, int n) =>
     IsGreen(attempt, target, n) ? SetChar(attempt, n, '*') : attempt;
```

This was, again, a conscious choice: I found that, especially when debugging, it was far clearer to read this function than to write e.g.

```
static string SetAttemptIfGreen(string attempt, string target, int n) =>
     IsGreen(attempt, target, n) ? SetChar(attempt, n, CharForInPlaceMatch) : attempt;
```

Especially as, in this system, that character is the one that is also used in the console user interface.

And the fact that functions have 100% unit test coverage, means that if I were to change this literal value in one place and overlook the others (the second reason for using named constants), my tests would immediately alert me.

I am well aware of, and generally support, the principle that named constants should be used in preference to literal values. However, coding style guides, like English style guides, are *guides* not hard-and-fast rules. All such guides may occasionally be broken in service of a higher goal.

Could the effectiveness of the Solver be improved?

Although I am extremely pleased with the results of the project, it is clear to me that the current implementation is not strictly *optimal*. The proper definition of the minimax algorithm is recursive: my current Wordle Solver optimises only the *next* attempt. That's a bit like writing a chess playing algorithm that looks only one move ahead (though including the opponent's immediate possible responses).

At present, Wordle Solver picks **RAISE** as the best first attempt, because the worst possible outcome (_ _ _ _ _ in this case) leaves only 167 possible words remaining, which is fewer than the worst outcome for any of the other 12,946 valid attempt words. If presented with this actual outcome, the solver always picks **BLUDY** (an archaic variant of the word 'bloody') as its next attempt, for which the worst outcome would then leave just 13 possible words. But might there be an alternative to **RAISE** that doesn't score quite so well on the first round, but which after *two* attempts never leaves more than 12 possible words, or perhaps even fewer?

Given that the current algorithm needs five attempts for only 3.6% of the possible puzzles, being able to guarantee to get every puzzle in *four* attempts seems tantalizingly close!

To guarantee to solve all Wordle puzzles in four or fewer attempts would mean that after *three* attempts there must be no more than *one* remaining possible word. To see if this were possible it would be necessary to run the algorithm three levels deep. The problem is combinatorial explosion. Although my Wordle solver can be shown to be of 'polynomial' complexity in (Big-O terms), with n being a large number (12,947 at present) even n^2 (i.e optimsation over the next *two* attempts deep) is going to be computationally very expensive.

Given that, even with the benefit of parallel processing, it takes 11 seconds to come up with **RAISE**, going just *two* levels deep will take several hours. Quite apart from the substantial coding effort involved, I don't fancy running my new laptop flat out at 100% processing usage for several hours! Modern microprocessors apparently have in-built thermal sensors and should 'throttle down' if in danger of overheating (thereby extending the required time further), but I'm none too confident about the impact of all that heat on the lithium batteries.

To re-purpose the best-known quotation from the movie *JAWS*, my advice to anyone wanting to take this project further is…

'You're gonna need a bigger computer!'

Appendix I: Lists of valid words and possible answer words

For this project I used a list of **2,309** possible answer words represented like this in the code:

```
public static List<string> AllPossibleAnswers =new List<string> {
"ABACK","ABASE","ABATE","ABBEY","ABBOT","ABHOR","ABIDE","ABLED","ABODE","ABORT"
```

to

```
"YOUNG","YOUTH","ZEBRA","ZESTY","ZONAL"});
```

For the list of **valid words** (i.e. for use as attempts) I created a new list made up from the 2,309 words above, plus an additional **10,638** words, totalling **12,947**:

```
public static List<string> ValidWords = new List<string> {
            ///First, all the Possible answers (repeated from above) then:

"AAHED","AALII","AARGH","AARTI","ABACA","ABACI","ABACS","ABAFT","ABAKA","ABAMP",
```

to

```
"ZUPAS","ZUPPA","ZURFS","ZUZIM","ZYGAL","ZYGON","ZYMES","ZYMIC"});
```

Note that it makes no difference to the Wordle Solver functions whether these lists are sorted into alphabetical order or not.

Both lists were obtained from the same source[9], however the first list (2,309) is available from many sources and widely claimed to be the list used in the Official Wordle game. (The full lists have not been included in this report in the interests of the environment, since it will be printed out.)

Addendum

Only after completing both the project and the report, I came across a GitHub repository[10] that lists some **14,855** words which it claims are valid attempt words from Wordle and, indeed, extracted from the Wordle source code. I haven't validated that claim, but I quickly ran the Wordle Solver again with this longer list – trying both variants of the algorithm in regular mode (it couldn't make any difference to running in Hard mode). The first variant still chose **RAISE**, but the second now chose a new word **OLATE** – which apparently means 'to lay waste'. Thankfully though, there was no significant difference in the results: it still took up to five guesses and, to the two decimal places already quoted, did no better on the average number of attempts: **3.55**.

[9] https://www.wordunscrambler.net/word-list/wordle-word-list

[10] https://gist.github.com/dracos/**dd0668f281e685bad51479e5acaadb93**