THE COMPUTER ALWAYS WINS

A PLAYFUL INTRODUCTION TO ALGORITHMS THROUGH PUZZLES AND STRATEGY GAMES

Elliot Lichtman

THE MIT PRESS
CAMBRIDGE, MASSACHUSETTS
LONDON, ENGLAND

PREFACE

Countless websites, books, and video courses promise to teach you to code. And many will. They will teach you vocabulary like PRINT, INPUT, and RANDOM. They will hammer home every comma, semicolon, and tab. But these courses routinely forget a simple truth: the joy of learning a new language comes in the wielding of it.

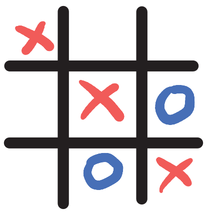
In this book, I therefore take a different approach. I assume that readers already know the bare-bones fundamentals of computer programming—the commands PRINT and INPUT; control structures that use IF, ELSE, FOR, and WHILE; plus the basics of numeric variables, string variables, lists, arrays, and functions—and, using only these foundational components, I immerse readers in the parts of computer science that are downright magical.

Come teach your computer to defeat human opponents in games like tic-tac-toe and Connect Four. Marvel at the strategies computers use to quickly solve puzzle games like Wordle and sudoku. Implement real machine learning where the computer plays a strategy, evaluates the result, and adjusts its approach based on that prior experience.

Do all that, and you will learn more than just how to code. You will learn, I hope, how to truly love coding.

Elliot

WHY ALGORITHMS?



If you are reading this book, you are almost certainly an unbeatable player at the classic game tic-tac-toe. Against an unsophisticated opponent, you win every time. Against a capable opponent, you might not win but you probably never lose. And you do all of this . . . how? What rules do you intuitively use to place each *X* or position each *O*?

To teach a computer to play, you would probably start by telling the computer to place its mark in any row, column, or diagonal where it already has two marks. That is, if the computer is in position to win on the next move, you would tell the computer to do so. Next, you would tell the computer to place its mark in any row, column, or diagonal where its opponent already controls two spaces. That is, if an opponent is in position to win on the next move, you would tell the computer to block. From there, there are a handful of plausible options, but perhaps you would tell the computer to place its mark in any row, column, or diagonal where it already has one mark and the other two spaces are empty. And you might tell the computer to, as a general rule, favor the center over the corners and the corners over any other open spaces.

If you did this, the truth is that your computer would play the game reasonably well. In fact, there are 2,097 theoretically possible tic-tac-toe gameboards, and these four simple rules will reliably identify the right move on 1,995 of them. No need for anything fancy. The computer can pick the best move in 95 percent of cases simply by adopting the strategy of winning if it can, blocking if it needs to, creating two-in-a-row combinations where the third space is blank, and, as a general default, favoring the center over the corners and the corners over any other space.

Teach your computer to play tic-tac-toe this way, however, and the computer will lose every game. Why? Because a good human player will exploit the 102 situations where this haphazard approach fails. At that point, no one will celebrate the fact that your computer would have done just fine in any of 1,995 other situations. It will matter only that your code has a blind spot, and your opponents can use that blind spot against you.

This leads to two important implications. First, the fact that intuitive rules fail for even simple games is the launching point for this book. The problem is that rules are like patches: they cover a specific situation, much like a simple patch can be used to cover a specific hole, but covering a large area by rule or patch will almost always leave accidental gaps. An effective computer algorithm must therefore be more like a blanket, covering the full range of possible situations, including those that are rare, hard to foresee, or just plain hard to articulate.

Second, simple rules, like patches, are nevertheless extremely valuable. In tic-tac-toe, for example, there is no better move than one that creates three in a row. So whenever that move is an option, a well-written computer program should cut short any more complicated algorithm and triumphantly take the win.

Intuitive rules, then, will serve an important role as we train the computer to play a wide variety of everyday games. But if the computer is to have any hope of winning against capable players, we will need something more. Exploring that something more is this book's exciting mission.

CHAPTER SUMMARIES

1 GUESS WRONG ANSWERS

You are playing Wordle, and you know that the hidden word ends with the letters *a-b-l-e*. Is *table* the best next guess? *Cable*? *Sable*? *Fable*? A well-trained computer will tell you not to guess any of these but instead to guess the seemingly nonsense word *scarf*. This chapter explains why, unpacking one of the best algorithms to use when your goal is to find the needle in some proverbial haystack.

2 THE ROAD NOT TAKEN

A foolproof way to escape a maze is to comprehensively test every path, keeping track of where you are and where you've been. You walk down a path, make note of every available option, and then continue forward until you either escape or hit a dead end. If you do hit a dead end, no problem; because you have kept track of all the remaining options, you can retrace your steps and try one of those. As it turns out, this is a promising approach for more than just mazes. If you are playing sudoku, for instance, there might be several numbers that could plausibly be placed in a given square. How to pick? Choose one tentatively, see if you can from there fill in the rest of the grid without hitting a dead end, and then retrace your steps if the original choice doesn't work. This chapter formalizes this wandering approach and then marvels at the countless puzzles it can effectively solve.

3 ONE STEP AT A TIME

If your car's navigation system applied the approach introduced in the previous chapter, you would from time to time be sent on a ridiculous journey. Your car would be parked in (say) Los Angeles, California; you would ask for directions to a nearby restaurant; and the computer would correctly tell you that one workable path would be to drive from California to Alaska, to Texas, to Ohio, and then back to that Los Angeles restaurant. Admittedly, you would ultimately arrive, but you would be hungry. This chapter therefore considers a competing algorithm that does not simply promise to find a winning path, but more powerfully promises to find the shortest one.

4 WHOSE TURN IS IT ANYWAY?

When you play any two-player game, you presumably pick your move based on what you think will happen in the moves to follow. In chess, for instance, as you think about moving your left-most pawn, you probably think ahead to what your opponent will do in response to that move, what you will do in response to your opponent's response, and so on, perhaps several moves deep. Computers can play games using this exact approach but with a huge advantage: while a human player will typically be exhausted after thinking just a few moves ahead, a computer can theoretically consider every possible future move.

5 MOVE FASTER

The prior chapter considers strategies where the computer picks its move by literally playing out every possible future response. That turns out to be a wildly effective but frustratingly slow approach. To exhaustively play out a game of tic-tac-toe, for example, the computer would need to test something on the order of half a million game interactions just to make its first move. Do the same thing for Connect Four and the computer is stuck evaluating untold trillions. And chess? Forget about it. This chapter takes a first step toward addressing this problem by limiting the extent to which the computer looks ahead. The computer might look two or three moves ahead, but in this chapter the computer is no longer allowed to play every possible game to its definitive conclusion.

6 PRUNING THE TREE

We can do even better, however. Our best algorithm so far still wastes a lot of time considering completely implausible moves. For instance, even if the computer realizes that its opponent will win the game unless the computer blocks a particular spot, our current algorithm records that information but then continues to consider other options. A human player would do nothing of the sort. Once a human player finds what looks to be the best move, the human player makes it. This chapter looks at strategies that empower the computer to similarly cut wasteful analysis, saving time for more difficult choices.

7 THROWING DARTS

When researchers are evaluating the efficacy of a medication, they don't test the drug on every patient. Instead, they test a few patients, then generalize the results to fit the population. Computers, it turns out, can do something very similar. For instance, instead of analyzing two potential moves rigorously, a computer can randomly simulate fifty games using the first option, randomly simulate fifty more games using the second option, and then pick the move with the better average performance. The power of this technique might surprise you, in that it can be both remarkably fast and surprisingly accurate.

8 AIMING DARTS

The prior chapter demonstrates the power of random sampling. This chapter takes the next step, shifting from random sampling to strategic sampling. For instance, if after fifty random simulations it is clear that one move is terrible while three others are plausible, a purely random approach would continue to explore all four options. A better strategy, however, would take those results and adjust, focusing all remaining simulations on the three still-plausible moves while ignoring the move that is pretty clearly a dud.

9 AIMING DARTS AT OTHERS

Let's not hurt anyone here, folks. But the two previous chapters apply random strategies to what are, in essence, one-player games. Here, we upgrade the algorithm so that it can be used to simulate two-player interactions. The chapter ends with a popcorn-worthy showdown: the two-player algorithms from chapters 4, 5, and 6 pitted against the two-player algorithms developed in chapters 7, 8, and 9.

10 ROCK, PAPER . . . PAPER

Random? Please. When playing rock-paper-scissors, you might try to make your choices randomly, but odds are you suffer from some idiosyncratic hiccup that throws off your game. Maybe you are reluctant to play scissors twice in a row, even though a purely random player would do exactly that surprisingly often. Maybe you subconsciously favor rock, or you absent-mindedly change your choice after a loss but keep it the same after a win. Because of this, rock-paper-scissors isn't really a game of random chance; it's a game of random chance plus pattern recognition. And who, dare I ask, is the king of pattern recognition? Yes, you guessed it: your computer.

11 BLACK BOXES

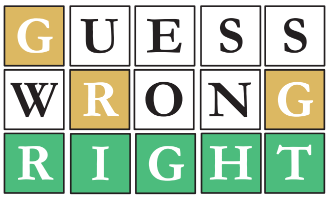
In every chapter thus far, I have been explicit about exactly what the computer is doing and why it works. At the cutting edge of computer science, however, are black-box strategies where these details are almost completely hidden from view. The programmer provides training data, which in our case will be some large number of already-played sample games. And the programmer builds flexible data structures and supportive functions that empower the computer to test different strategies against the data. But from there the computer finds its own way, learning from the past to create its own strategic future.

12 MINIMIZING REGRET

People learn by interacting with their environment. Toddlers, for instance, figure out the details of walking not by listening to detailed instructions from their caregivers but by paying attention to their own bumps and bruises. Computers, too, can learn as they go. Thus, this chapter concludes our work by exploring one such learning algorithm: the computer plays the game, looks back to see where it might have made a better move, and quantifies that regret in order to gradually develop an even more promising solution strategy.

SEARCHING AND SORTING

1
GUESS WRONG ANSWERS



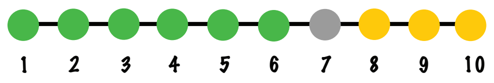
I am thinking of a number from 1 to 10. I chose my number randomly, and when you guess I will honestly tell you whether your guess is spot-on, too high, or too low. What should your first guess be?

Because I chose my number randomly, you might think that any first guess is equally good. If you guess the number 2, for example, you have a 1-in-10 chance of being correct. If you guess the number 9, you again have a 1-in-10 chance of being correct. Indeed, as long as you guess a number between 1 and 10, you might think that any guess is just about the same.

But no.

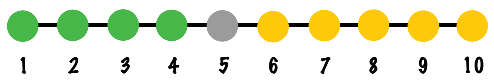
Because I will also be telling you whether your guess is too high or too low, different guesses have very different implications. Take an extreme: Suppose you guess the number 1. Make that guess, and you have a 1-in-10 chance of winning the game and a 9-in-10 chance of being left to choose from among nine numbers, all of which are greater than 1. Compare that with a guess of 7. You would still have a 1-in-10 chance of being right, but now you would also have a 6-in-10 chance of being told that your guess is too high, and a 3-in-10 chance of being told that your guess is too low.

Either way, you would learn a ton of information about the right answer. If your guess turns out to be too high, you can suddenly eliminate from contention the numbers 7, 8, 9, and 10. Too low, and (even better) you can eliminate seven numbers, namely 1, 2, 3, 4, 5, 6, and 7. A guess of



the number 7 is thus tremendously helpful even if it turns out to be wrong, in that it serves to eliminate a huge number of incorrect answers, making your next guess that much more likely to be right.

And 7 is not even the optimal pick. As noted above, by guessing the number 7, you give yourself a 6-in-10 chance of cutting the list down to six numbers and a 3-in-10 chance of cutting the list down to three. An incorrect guess of 7 thus leaves you to choose from 4.5 numbers on average. You can do even better, however, by guessing either of the two numbers in the middle of the range, namely the numbers 5 or 6. Try 5, for instance. Once more, you would have a 1-in-10 chance of being right, but this time you would have a 4-in-10 chance of being too high and a 5-in-10 chance of being too low. The first of those possibilities would eliminate six numbers and leave you with four. The second of those possibilities would eliminate five numbers and leave you with five. Adding and multiplying as appropriate, an incorrect guess of 5 would leave you with just 4.1 options on average, an outcome significantly better than the 4.5 associated with an incorrect guess of 7. Guessing 6 is then an equally good option, for essentially the same reasons.



We can think of algorithms like this as *elimination* algorithms, where the key insight is that these strategies focus not solely on picking right answers but also on quickly eliminating wrong ones. And elimination is way more powerful than you think. Play our guessing game using the numbers 1 through 1,024, for example, and an algorithm that always picks the middle number guarantees you a win after no more than ten guesses. Play the game using numbers up to 1,000,000, and pick-the-middle guarantees you a win within twenty tries. The underlying math in these examples is just division. Every time you pick the middle number, either you win or you eliminate half the remaining options.

$$1{,}024 \rightarrow 512 \rightarrow 256 \rightarrow 128 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

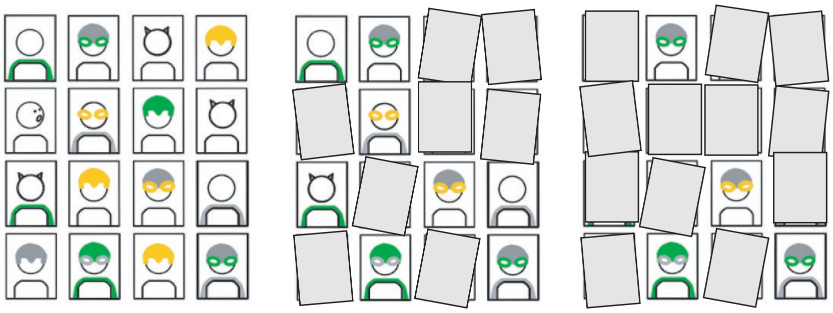
Players using a guess-the-middle strategy cut the list of numbers in half every move. Starting with 1,024 numbers, then, a player's worst-case scenario is to have 512 numbers left after the first guess, 256 left after the second guess, and just 1 left after the tenth guess.

If you are with me thus far, you are probably beginning to realize that one of the games you played as a kid was a lot more interesting than you appreciated at the time. The game I have in mind: Guess Who?

Here's how the game works. You are given a gameboard with some number of cartoon faces and your opponent chooses one of them as their secret character. Your opponent is also given a gameboard with some number of cartoon faces, and you, too, choose one to be your secret character. From there, the two of you take turns asking each other yes/no questions, racing to be first to identify the other person's mystery pick. I had the superhero version of the game, so as a kid I would ask questions like "Does your secret hero wear a cape?" or "Can you see your secret hero's hair?"

What's interesting is that, as a kid, I always asked straightforward questions like the examples given above, and I was pleased whenever an affirmative answer would eliminate, say, four or five heroes. But we just saw that dividing by two is the fastest way to whittle down a list, which means I should have been asking questions designed to create two equal-sized groups every time. How? One effective approach would have been to sharpen my questions using words like *and* and *or*.

Against the sample board shown below, for instance, a great opening question would be to ask whether the secret hero has either a cape *or* goggles, since half the characters satisfy that constraint but half do not. A positive response might best be followed by another compound question, maybe asking whether the secret hero has both a helmet *and* goggles.



For the sample gameboard on the left, a question about "capes or goggles" eliminates half the faces. From there, a follow-up about "helmets and goggles" would again divide the number of remaining faces by two.

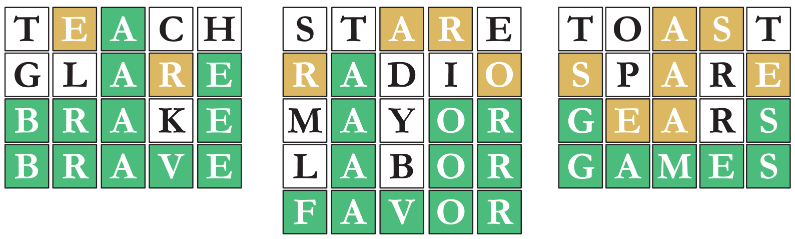
YOUR MOVE

Suppose you are playing this divide-by-two strategy against a player who asks conventional questions, and that player gets lucky: they ask a ridiculously lopsided question like "Does your secret hero have horns?" at a time when the hidden hero does, in fact, sport two giant protrusions. What should you do next? Should you stick with your clever *and/or* approach? Should you instead try your own lopsided question in the hope of catching up? And does this mean that sophisticated strategies work better against sophisticated players than they do against unsophisticated players?

The games we have considered thus far have all been games in which it was relatively easy to implement elimination. In guess-a-number, for instance, we had to keep track of an ordered list of numbers, and we had to find the middle number in that list. In Guess Who, we had to keep track of which cartoon characters remained in contention and also notice relevant characteristics that could differentiate the various characters. But elimination algorithms sometimes require so much recordkeeping that no human player can possibly track all the data, let alone identify the optimal move. It is in those instances that computers can meaningfully outperform their human counterparts.

Consider the game Wordle in this light. A five-letter word is chosen at random and hidden from the player. The player then attempts to guess the hidden word but with one important constraint: every guess must itself be a real five-letter word. After each guess, the player is given feedback about the letters they chose. If a letter in the guess is highlighted green, that letter is not only found in the hidden word but also found at exactly the same position in that word as it is in the guessed word. If a letter is highlighted yellow, the letter is in the hidden word but at some other spot. Lastly, if a letter is neither green nor yellow, the letter is not part of the hidden word at all.

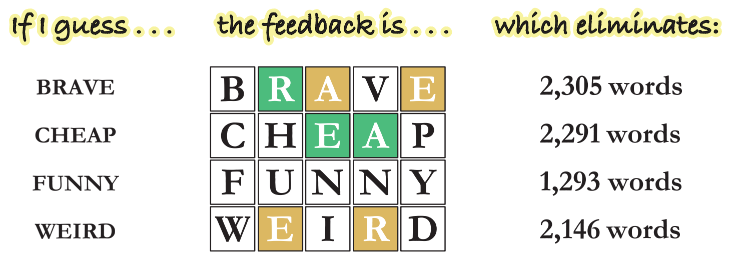
Three sample games are shown below. For instance, in the first game on the left, my opening guess was the word *teach*. I earned a close-but-not-quite reaction for the letter *e* and a full-throated endorsement for my placement of the letter *a*, so I knew that the hidden word uses the letter *e*, has an *a* in the middle, and does not use the letters *t*, *c,* or *h*. That led to my second guess, the word *glare*, which turns out to correctly place both the *e* and the *a* and then also reveal, thanks to the new shaded square, that the hidden word has an *r* in either the first or second position. My third guess of *brake* was then very helpful; it taught me that the first letter of the hidden word is *b* and that *r* is in fact the second letter. From there, there was only one plausible option left to guess, so I chose *brave* and won the game in that fourth move.



In the first and third games, I was able to guess the hidden word on my fourth try. In the second game, I came close but needed a fifth guess to finally get there.

One intuitive way to play Wordle is to focus early guesses on letters that tend to show up frequently in five-letter words. Using this approach, a player might open the game by guessing the word *raise* because the letters *r*, *a*, *i*, *s*, and *e* each commonly appear in real five-letter English words. A more sophisticated approach might consider letter position, too, perhaps picking a word like *raves* because the letters *a* and *s* are not only commonly used in five-letter words but also frequently appear in the second and final positions, respectively. A player adopting this approach would then piece together the hidden word letter by letter. For instance, if an opening guess of *beast* gave favorable information about the placement of the letter *e* and the inclusion of the letter *t*, the player might try, as their next guess, a word like *tempo* or *fetch*, two words that keep the *e* in the second spot and have a *t* somewhere other than the second and fifth positions.

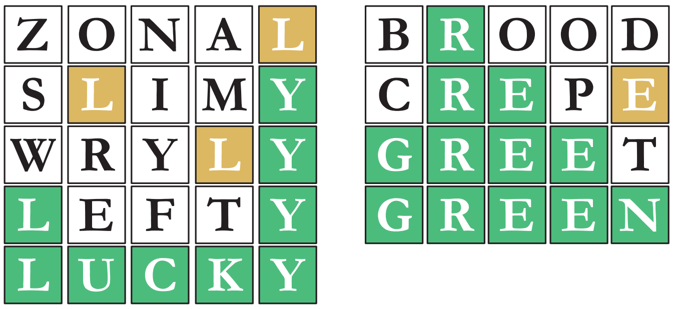
But now try an elimination approach. The *New York Times* publishes new Wordle puzzles every day, choosing hidden words from a database of 2,315 five-letter options. In an elimination strategy, the goal would be to eliminate some substantial number of those 2,315 words in the first guess. For example, if we were to pick the word *apple* and end up with no green and no yellow boxes, that feedback would turn out to eliminate 1,888 words from the list: every word that uses an *a*, a *p*, an *l*, or an *e*. If we pick the word *apple* and end up with a green *l* and a yellow *e*, the feedback this time would eliminate an even more impressive 2,297 words: every word that uses an *a*, every word that uses a *p*, every word that has a letter other than *l* in the fourth position, and every word made without even one *e*.



If I guess . . .	the feedback is . . .	which eliminates:
BRAVE	⬜B 🟩R 🟨A ⬜V 🟨E	2,305 words
CHEAP	⬜C ⬜H 🟩E 🟩A ⬜P	2,291 words
FUNNY	⬜F ⬜U ⬜N ⬜N ⬜Y	1,293 words
WEIRD	⬜W 🟨E ⬜I 🟨R ⬜D	2,146 words

Even random guesses can quickly eliminate large numbers of previously possible words. The feedback from a guess of *brave*, for example, makes clear that 2,305 words in the word bank are not right because those words would have led to different feedback.

The chart above captures four additional examples, comparing the random guesses *brave*, *cheap*, *funny,* and *weird* against the hypothetical hidden word *great*. The promising payoff: nearly every guess eliminates hundreds of potential words, making the next guess that much easier. This suggests a plausible Wordle algorithm. We would begin with the full *New York Times* list of all eligible Wordle words. The computer would then guess one of those words at random. Based on the feedback, the computer would eliminate the clear losers and then draw a new random guess from among the remaining words. The computer would repeat that process again and again until the hidden word was either proposed as a guess or was the only word left in the queue. And that would work fine. Indeed, when I tested this approach, the computer guessed my hidden word *lucky* in just five tries, and it guessed my hidden word *green* in four. I then tested the entire 2,315-word database, and the random approach was able to identify the hidden word in fewer than five guesses roughly one third of the time, and it needed fewer than seven guesses for all but 187 words. That's not genius performance, but it's not bad.



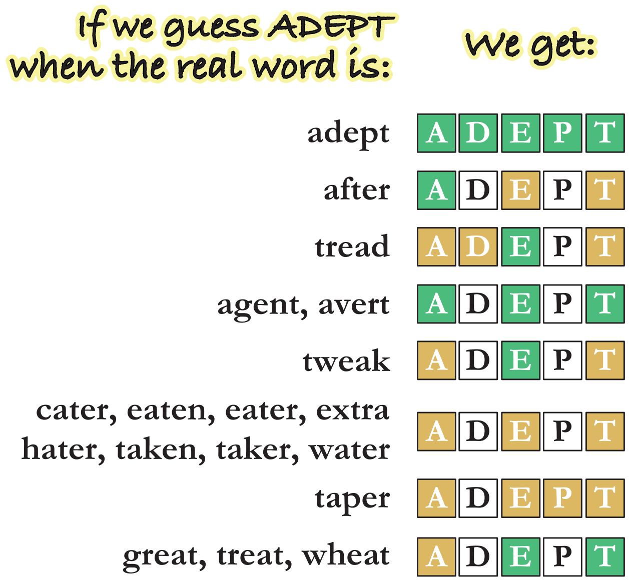
When my hidden word was *lucky*, the computer's random guesses were, in order, *zonal*, *slimy*, *wryly*, *lefty,* and *lucky*. When my hidden word was *green*, the computer guessed *brood*, then *crepe*, then *greet*, then *green*.

But as we learned from both guess-a-number and Guess Who, computers do even better when they pick their guesses strategically. To consider how to be strategic in this context, let's play the game with an artificially short list of permissible words so that we can really see what happens as the computer makes different guesses. Thus, instead of using all 2,315 words that the *New York Times* itself uses to create its daily puzzle, consider a simplified version of the game where the hidden word is guaranteed to be one of the eighteen words listed below.

adept, after, agent, avert, cater, eaten, eater, extra, hater, taken, taker, water, great, treat, wheat, taper, tread, tweak

If our first guess is the word *adept* and the hidden word is also *adept*, we will obviously win the game. Assuming that the hidden word is picked at random, there is a 1-in-18 chance of that happening. By the same logic, there is a 1-in-18 chance that the hidden word is *after*, a 1-in-18 chance that it is *agent*, a 1-in-18 chance that it is *actor*, and so on, for all eighteen words. The chart below captures the feedback we should therefore expect. For example, if we guess *adept* and the hidden word is *great*, *treat*, or *wheat*, the feedback we will receive will be that the *e* and *t* are both properly placed and the *a* is correct but in the wrong position. There is a 3-in-18 chance of that happening—the chance that the hidden word is *great*, *treat*, or *wheat*—and, if we do get that feedback, in our next guess we would need to distinguish between only those three words.

By the same logic, if we guess *adept* and the hidden word is *cater*, *eaten*, *eater*, *extra*, *hater*, *taken*, *taker*, or *water*, we will see a response indicating that the *a*, *e,* and *t* are all in the puzzle but in other positions. There are eight ways that can happen, and, if it does, in our next move we will have to distinguish between only those eight possible words. The chart below runs through the full list in this spirit. If we guess *adept*, these are all the possible responses we might see, and what those responses mean in terms of how many words would still be in contention.



This chart shows the full range of feedback we might see if we guess the word *adept*. Some of the feedback immediately makes clear what the hidden word must be. Other feedback is less conclusive but still helps to significantly narrow the list of plausible words.

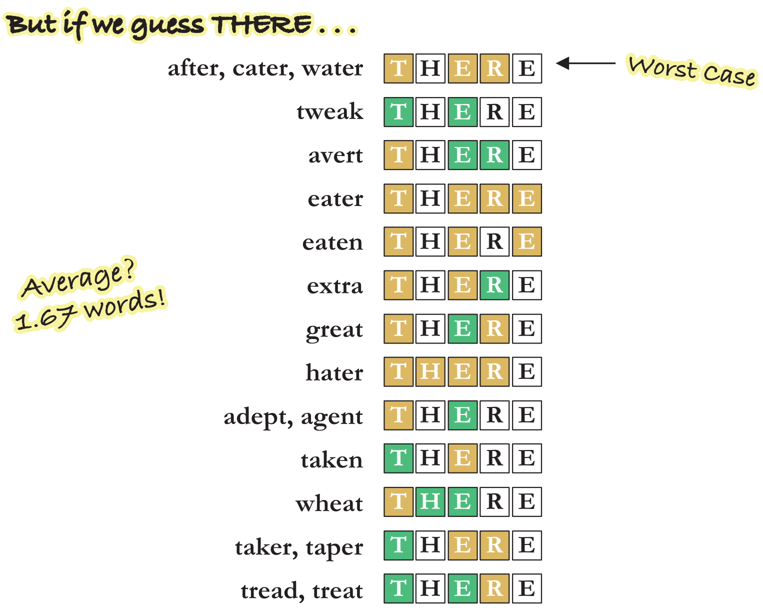
We can make similar charts for all eighteen words in the list. Consider, for example, a chart focused on the guess *extra*. If the feedback were to indicate that the *e*, *t*, and *a* are all used in the hidden word but in other locations, that would allow us to narrow the list of remaining words to just five: *adept*, *agent*, *taken*, *tweak*, and *wheat*. Were we instead to receive feedback indicating that the *e* and *t* are both correctly placed while the *a* is part of the hidden word but located elsewhere, we would immediately know that the hidden word must be *eaten* because that is the only word on the list that satisfies those constraints and does not include either an *x* or an *r*.

If we made all eighteen charts, we could then pick the optimal guess. Consider the word *taken*. For *adept*, the worst outcome we found was feedback that left us with eight remaining words. For a guess of *taken*, by contrast, the worst case turns out to be feedback that endorses the use of the letters *t*, *a*, and *e*, but in other positions, which leaves five words in play: *adept*, *avert*, *extra*, *great*, and *wheat*. If our goal is to minimize our worst case, *taken* is thus a better guess than *adept* because its worst case leaves us with fewer words. Even better options along these lines are the words *after*, *cater*, *eater*, *hater*, *taker*, *taper*, and *water* because, for each of those, the worst case leaves us with only four possible words.

Instead of focusing on the worst case, we could instead focus on the average number of words expected to remain after an incorrect guess. For *adept*, we had an 8-out-of-18 chance of being left with eight words, a 3-out-of-18 chance of being left with three words, a 2-out-of-18 chance of being left with two words, and a 4-out-of-18 chance of being left with just one word. Adding and multiplying as appropriate, this suggests that we would, on average, be left with 4.5 words after wrongly guessing *adept*. Running the same math for *taken* tells us that, on average, an incorrect guess of *taken* will leave us with 2.83 words to consider. And running this same calculation for all eighteen words in the list ends up suggesting that our best bets are the words *water* and *taper* because each of those leaves us with an average of merely 2.17 words for our next guess.

But hold on. Ready for this? Our analysis thus far has only considered guesses that could turn out to be the actual hidden word. That is, we have eighteen words in this version of the game and we have only considered using those eighteen words as possible guesses. But crazy as it might sound, it is possible that some other word will be an even better guess. For example, consider the word *there*. Because *there* is not in our list of eighteen possible words, *there* will definitely not be the right answer. However, as the chart below shows, the worst-case outcome associated with the word *there* would be a list of three remaining words, and the average number of words remaining would be a shockingly low 1.67. This means that *there*—a word not in our list of possible answers—is likely a better guess than *adept*, *taken*, or indeed any word in our list, regardless of whether our goal is to minimize the worst case or to minimize the average number of words remaining.

This is surprising. An intuitive human player would never guess *there* because *there* is not a possible winning word in this example. But in an elimination algorithm, part of our goal is to eliminate as many duds as we can, and so all of a sudden a word like *there* might well be our best option even though we know we cannot possibly win the game by guessing that particular word.



The word *there* cannot be the correct answer because it is not listed in the word bank. As a first guess, however, it turns out to be a fabulous choice regardless of whether the goal is to minimize the worst case or minimize the average.

CHAPTER CHALLENGE

Your challenge this chapter is to write a Wordle solver using the elimination algorithm we just explored. To get you started, this CodeLink will take you to sample Python code that plays a simple version of the game. There is an array called wordList that stores a few thousand permissible five-letter words. There is a function called hideWord() that randomly chooses one of the words to be the hidden word; a function called guessWord() that allows the computer to guess; and a function called scoreGuess() that evaluates the guess and provides feedback on the screen. In the sample code, however, the computer's guesses are completely random. That is, the computer randomly chooses a word from the list, guesses it, and, if that word is not the hidden word, the computer randomly chooses some other word, having learned nothing from its prior tries. Your job is to replace this random function with a function that implements a thoughtful elimination process, perhaps based on the average number of words remaining or on worst-case analysis.

As you code, think about whether there are other improvements you can make to the algorithm. For example, suppose that guessing the word *mouse* would leave the words *catch*, *match*, and *latch* as the remaining possible choices, whereas guessing the word *party* would leave the words *north*, *first*, and *aargh*. In our work thus far, we have treated these two outcomes as equivalent because in both cases there are three words left to consider. But *catch*, *match*, and *latch* are worse words than *north*, *first*, and *aargh* because it is more difficult in the next guess to distinguish between *catch*, *match*, and *latch* than it would be to distinguish between *north*, *first*, and *aargh*. Is there a plausible improvement to our algorithm that would account for the fact that some lists of words are actually easier to evaluate than others?

Lastly, if you enjoy Wordle, consider how you might change the algorithm to play other versions of the game. For instance, in Fibble, the letter-by-letter feedback includes one lie per round, such that a player might be told that the letter *f* is in the right spot when, in fact, *f* is not used in the hidden word at all. Or, my favorite Wordle variation: in Absurdle, the secret word changes from round to round. The new word is always consistent with the feedback already given, but beyond that the word is chosen to make things as hard as possible for the guessing player.