- I'm Adam Forsyth
- Group Engineering Lead & Community Lead @ Braintree
- This talk is called Beating Mastermind
- Code, slides, and speaker notes will be posted on my github

Beating Mastermind

Winning Games, Translating Math to Code, and Learning from Donald Knuth

Adam Forsyth

github.com/agfor

- This talk is about a game called Mastermind
- But really it's about being comfortable with algorithms
- And about how to take an academic paper describing an algorithm
- And turn it into code
- Also, a little bit about an algorithm called minimax in particular

- So what is Mastermind? It's a two-player codebreaking board game
- This is what the one I had as a child looked like
- One player, the codemaker, makes up a pattern of colored pegs, keeps it hidden
- Other player, the codebreaker, guesses it, getting feedback in the form of B/W pegs after each guess
- Based on a century-old pencil and paper game called Bulls and Cows

- Here's an example game, walk you through it.
- The secret pattern is at the bottom, hidden until game is over (and off the screen here).
- Guesses start from the top
- First guess, no circles to the right, so no reds in the answer
- Second guess, two black circles on the right. Means two of the guesses are the right color in the right position. So still need to figure out which two.

- Third guess, one black and one white -- meaning one yellow is in the wrong position, but one is correct, greens are wrong.
- Fourth guess, both yellows are correct and one blue
- Fifth guess, blue in wrong position, orange wrong. Sixth guess, I win!

- The screenshot I just showed is from Simon Tatham's Portable Puzzle Collection
- Amazing collection of logic puzzles downloadable or playable online
- Seeing Mastermind (called "Guess") here is what inspired this work and talk
- Wanted to include more screenshots of the game illustrating the steps of the algorithm, but unfortunately had to cut them due to time.

- Now let's talk about the paper. It was written by Donald Knuth, published in JoRM in 1976
- Known as the "Father of analysis of algorithms", created TeX typesetting system, wrote TAoCP
- Gives a solution that will always win in 5 moves or less

- The problem is that the paper gives the solution as a big lookup table called figure 1 (this is only the first half)
- Does include a brief description of how to use it.
- But that doesn't show why / how it works, and there is no code.
- If you didn't care, you could copy the lookup table into your program and we could stop the talk there
- But we want to understand it!

- The paper basically has 7 parts. READ 1-3.
- In the examples it uses the numbers 1-6 instead of colored pegs, and "B" or "W" instead of black or white pegs.
- The 4th part is more interesting. It describes an important part of why the algorithm picked a certain guess in the previous example.

- This is part 4. READ IT
- What this is saying is that sometimes, we'll need to guess combinations of colors that we know are impossible, in order to eliminate enough potential answers that it is possible to guarantee a win in only five moves.
- This is pretty cool. That's something a human player would be unlikely to do intuitively, and I'd never intentionally done when playing the game before reading this paper.

- This is part 5. READ IT
- This is the heart of the paper. To restate, you can get one of 15 different combinations of black and white pegs as feedback to your guess, depending on how accurate it is. You want to pick a guess that *most* reduces the number of possible correct answers left, assuming you get the feedback that gives you the *least* amount of information
- In other words, you want to minimize the maximum number of remaining potential answers

- It then describes how to break ties
- If there are multiple guesses that leave the same number of remaining potential answers, choose the one that is itself a possible answer.
- If there are still more than one, sort them numerically -- remember we're using the numbers 1-6 here instead of colors.

- So now we need to figure out, how do we implement that?
- The answer, of course, is to Google it
- The fifth result is a Wikipedia article titled "Minimax"
- For those of you out there who are Stack Overflow fans, don't worry -- several of the other top answers are Stack Overflow.

- I'm not going to go into the exact definition of Minimax
- It's pretty much what we've just described
- Read the Wikipedia article if you want to know more
- Since we're impatient programmers, we're going to skip down to the pseudocode, section 2.2, and hope it will point us in the right direction

- So this is pretty short, that's promising. Let's walk through it, starting with the function parameters.
- The node is where we are in the game -- what possibilities we've already eliminated, our guess, and what score it got.
- The depth is how much further ahead in the game we want to look to see which guess is best
- If you go back and read part five in the paper, it says to look at which is best based on the possible responses to your next guess by the codemaker -- so that's only one guess and one response ahead. So our depth is two.

- Now let's look at the second and third lines.
- If the depth is zero, we've already looked ahead and we need to provide a value for the guess
- In our case we'll never reach a terminal node before reaching zero depth.
- Remember from part five of the paper, the heuristic for a score from the codemaker is the number of possibilities it leaves
- And the heuristic for a guess is that number, whether the guess itself is still possible, and then finally pick numerically, using 1-6 instead of colors.

- Now let's look at the middle part.
- Since we, as the codebreaker, want to minimize the maximum, we're the minimizing player, and the codemaker is the maximizing player.
- The child nodes are the possible scores a guess could get
- So for the maximizing player, we look at all possible scores and take the maximum heuristic

- Now let's look at the last few lines.
- Again, the child nodes are the possible guesses we could make
- And the minimizing player is the codebreaker
- So we take the minimum out of the heuristic values returned by the next depth down, the maximizing player.
- Starting from the first call to the function, we first look at each guess and call the function again, then for each one we look at each score it could have, and call the function a third time, then we calculate a heuristic value, go back out a call and take the maximum, and go out one more call and take the minimum, and that's our result.

- So at this point, we've done the hard work
- We've read the paper and found the description of how it solved the problem
- Googled that, and found pseudocode of how to implement that kind of algorithm
- Now we need to code it! Specifically, I'm going to show an implementation of Mastermind that automatically plays against a randomly-generated answer.
- Don't worry if you don't follow the code exactly, you can always download it later and play with it yourself.
- This code could also be structured better, but I've made it shorter and simpler in some ways for the purpose of this talk. You've been warned.
- All of this boils down to about 60 lines of actual code.

- First I'm going to show you the basic logic needed for Mastermind
- When we start a new game, we need to set up a few things.
- We haven't made any guesses yet, and we select a random answer to check against.
- We also setup lists of all possible answers and the map of all possible guess / answer combinations to their corresponding scores. I'll show you that in more detail in a minute.

- Here's the man loop of the game. You can make up to 10 guesses.
- We check that each guess is in the set of all possible answers.
- If it's valid, we increment the number of guesses used, and calculate a score for the guess, which I'll show in detail next.
- Finally, if our score was four black pegs, we've won! We print out how many guesses it took us and the answer, then break out of the loop.
- I don't handle losing here to keep the code short, since our algorithm always wins.

- Here's the function that takes a guess and an answer and calculates a score for them.
- We make three empty lists, one to store the score as we build it up, one to store the list of pegs from our guess that don't match the answer, and one to store the list of pegs in the answer that don't match the guess.
- Then we go through each pair of pegs that are in the same location, and for each one that matches, we add a black peg to the score. For each pair that doesn't match, we store them for later.
- Then we go through the pegs from the guess that didn't match an answer peg in the same location, and see if they matched one of the left over pegs from the answer that were in another location.
- If they match, we add a white peg to the score, and remove that answer peg from the list to consider for more white pegs.

- Before we use the minimax algorithm, there is some information we need.
- It's a little slow to build up this information, so we do it once in the `__init__` method, so we can use it repeatedly if we play more than once. The rest of the algorithm is fast.
- First, we need to generate all possible answers, which means all possible combinations of the digits 1-6.
- Second, we use that list to generate all possible combinations of guess and answer, and for each one, calculate the score we'd get back from the codemaker for that combo.

- Finally, the climax. The minimax algorithm, using the heuristic from the paper, implemented in Python.
- Initially, we're looking at only when we've already made a guess -- not the first move. That's because we need to already have a score and have eliminated some possibilities for our heuristic to effectively differentiate between different potential guesses.
- We look at the list of possible answers, and for each one, we check whether it's still possible, given our last guess and the score it got. We discard it if it's no longer possible.
- We also create an empty list that we're going to use to store the heuristic values from the algorithm for each potential guess.

- We then go through every possible guess, along with its set of scores and the answers they correspond to, that we set up at the beginning. This is our first call to the function, for the minimizing player / codebreaker.
- For each one, we filter the dictionary of possible answers and their scores just like we did previously -- making sure they're still possible. We only filter answers, not guesses, since we may need to make a guess that isn't still possible to win in only five moves. This is the second call to the function, going through the scores we could get from the maximizing player / codemaker.
- We then store that new dictionary for later use

- Next, we count how many times each score appears among the potential answers for the given guess. This tells us how many potential answers would be left for each score we could possibly get from the codemaker for our guess. This is the third and innermost call to the function, calculating the heuristics.
- We then look for the maximum number of potential answers we counted -- this is our worst-case scenario for that guess.

- Now let's look at the last two lines in this snippet
- Then remember how we break ties if two possible guesses have the same worst-case number of potential answers. We want to prefer guesses that are in the list of possible answers, so we check that for the guess.
- Finally, we append the combined heuristic, made up of the worst-case number of potential answers, whether or not the guess is itself a possible answer, and the guess itself (since its numeric value is the final tiebreaker) to the list of heuristic values.
- We've now completed the second call to the function, for the maximizing player / codemaker.

- There are a couple of lines left. First let's look at the bottom of the snippet.
- After we've built up our list of heuristic values, we take the minimum one -- the minimum, maximum, number of potential answers, with ties broken by whether the guess is a potential answer, and finally with remaining ties broken by the value of the guess itself. This completes the pseudocode we saw on Wikipedia.
- We pull the guess itself back out of the heuristic value, since that's what we're looking to get out of this function, by using `[-1]` to get the last item out of the tuple.
- Finally, look back at the top, and remember we're inside an `if` statement. We were only doing all of that if we'd already previously made a guess.
- If we haven't already made a guess, then for our first guess we use "1122", as you can see at the bottom. It turns out that starting with this combination, two of one color and two of another, is the only starting guess that always lets us win in only five moves, which is explained near the end of the paper.

- So that's it -- you call the "play" method and the game will play itself.
- Run the code for you in a loop, so you can see it takes a few seconds to start up, and then less than a second to win the game each time -- and always in five moves or less.

- That's all I have
- Code and slides with speaker notes on github
- github.com/agfor
- Questions?