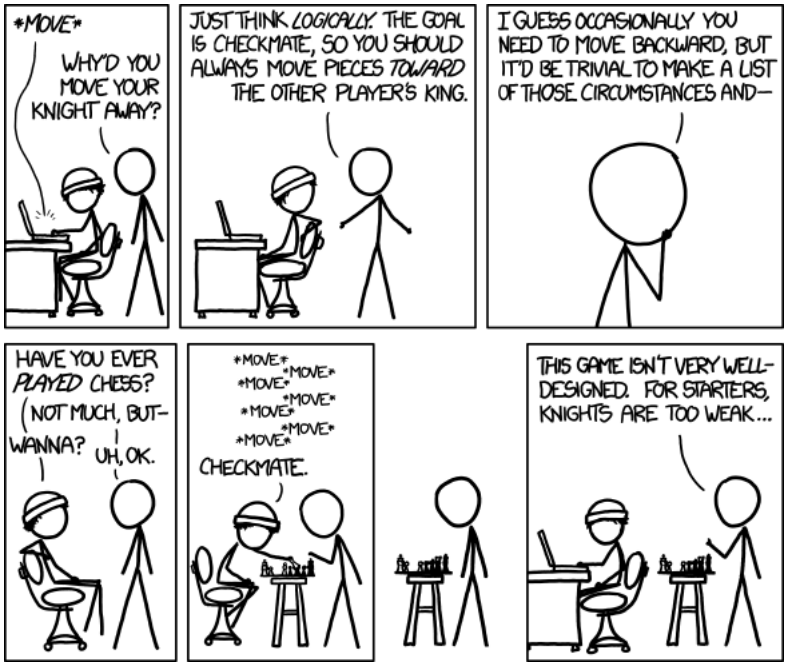
Computer Science 52

Assignment 9

Due Friday, December 1, 2017, at 5:00 PM

Important note on scheduling Because of the intervening Thanksgiving Recess, we have a bit of overlap on Assignments 9 and 10. The former would normally be due on November 24, the day after Thanksgiving. We are giving you a little extra flexibility and making it due a week later. However, Assignment 10 is will be due one week later, on December 8. We strongly encourage you to finish Assignment 9 early and move on to Assignment 10!

Researchers in artificial intelligence and algorithms often use games to explore computational strategies. Recent news reports identify successful programs that play sophisticated games, like chess, Go, and Jeopardy. There is also a large body of work on simpler games, like tic-tac-toe, Nim, and Mastermind. The techniques may involve careful mathematical analysis, logic programming, genetic algorithms, and when feasible, exhaustive computation. In this assignment, you will have the opportunity to exercise your skills with SML, recursion, lists, and other data structures and you create programs to play Mastermind.



https://xkcd.com/1112/

Submission: As usual, obtain the template and check files from the course Calendar page. Rename the template file to `asgt09.sml` and put your solutions in the marked places. When you are done, verify your work with the check file. Submit your file on the course submission page.

Mastermind, revisited

Recall from Assignment 3 that Mastermind is a game for two players, a codemaker and a codebreaker. The codemaker chooses a secret code, an arrangement of four colored pegs. On each turn the codebreaker presents a test arrangement, and the codemaker responds with the number of *exact matches* (in which a color appears in the same place as in the secret code) and the number of *inexact matches* (in which a color appears in the secret code, but not in the same place). The codebreaker's objective is to discover the secret code in the smallest number of rounds. Each round gives the codebreaker new information by narrowing the set of possibilities for the secret code.

In Assignment 3 we saw a crude strategy for playing the Mastermind. Here we will adopt a better strategy, published by Donald Knuth in 1976. His paper, linked on the course Resources page, is short and accessible; consult it for details on the strategy. Knuth shows—for the usual six-color, four-peg version of Mastermind—that it is possible to guarantee a win in five rounds but not fewer. The expected number of rounds using the strategy is 4.476. Those values are significantly better than the maximum of nine and the expected number of 5.765 for our crude strategy in Assignment 3.

Donald E. Knuth. The computer as Master Mind. *Journal of Recreational Mathematics,* 9(1):1–6, 1976.

Our strategy in Assignment 3 was to maintain a list of possible solutions and to use the first element of that list as the guess for the next round. Rather than take the first element of the list, Knuth's strategy is to use a "good guess," one that will yield the smallest number of remaining possibilities. It is, of course, impossible to compute the "best guess"; it is the secret code itself. So we settle for the "least bad guess." Specifically, we score each possible guess by computing the maximum number, over all the responses that the codemaker might give, of the remaining possibilities. Then we choose the guess with the smallest score.

In game theory, this idea is called a *minimax* strategy.

As you will soon see, the strategy is computationally intensive. It is time-consuming to carry out the computation "on the fly," as we did in Assignment 3. Instead, we will pre-compute the strategy and store it as a tree data structure. The tree will tell us the next guess and, based on the codemaker's response, what strategy to employ next. At each

node the tree will have a branch for each of the possible responses by the codemaker. Each of the branches will itself be a strategy, so we can know what move to make next.

```
datatype peg = Red | Orange | Yellow | Green | Blue | Violet |
               Magenta | Turquoise | Celadon;

type code = peg list;

type response = int * int;

type codemaker = code -> response;

datatype knuth_tree = Lose
                    | Step of code * knuth_tree list
                    | Win;
```

Here we have introduced types to make our declarations more descriptive. They are merely abbreviations, and SML requires a bit of coercion to display them. In a few places (very few, actually) you will have to specify the types, as in
`fun matches (secret:code) (guess:code) = ...`

The heart of our work is the `Step` part of the `knuth_tree`. It tells the codebreaker which `code` to play and provides a collection of trees, one for each possible response by the codemaker, to guide the next round of play. Obviously, the game is over when the tree is `Win` or `Lose`.

"Lose" is perhaps not the best term. The `Lose` value is not possible if we really have a winning strategy. Something like `NotPossible` might be a better term, but we opt for the shorter word.

We will have to keep in mind that our datatypes are more limited than their declarations indicate. A `code` will be a fixed-length list, size four in the typical board game but perhaps longer in some of our experiments. A `response` will be a pair of *small* integers—the number of exact and inexact matches will be non-negative, and they must add to a value no greater than the length of the codes.

Some of our work on Assignment 3 is useful here. In particular, two functions from Assignment 3 appear in the template file.

```
matches : code -> code -> response
isConsistent : code -> response -> code -> bool
```

Variations on Mastermind

You will be working with several variations on the basic game. The game (C, P)-Mastermind is the game in which there are C possible colors and P pegs in a code. The standard game that we have been discussing is $(6, 4)$-Mastermind. You may want to experiment with $(2, 1)$-Mastermind (utterly trivial) on up to much larger games. The standard game, and the ones beyond it, are computationally intensive, so you will want to restrict attention to some of the smaller games as you are working on your code.

Here is a declaration from the top of the template file. In this case we have $(3, 3)$-Mastermind.

```
val (colorCount, pegCount) = (3,3);
```

You can configure the size of the game by specifying the number of colors and pegs at the beginning. Both numbers must be positive integers, and `colorCount` cannot be greater than nine. A block of code later in the template file sets some important parameters.

- `allColors` is a list of the colors currently in use.
- `allCodes` is a list of all the possible codes, using the colors and number of pegs specified.
- `allResponses` is a list of the possible responses, based on the number of pegs. The list of `knuth_trees` in a `Step` is ordered according to the order of elements in `allResponses`.
- `winningResponse` is the response that says the codebreaker has won the game.

Use these values throughout your code. It should never be necessary to name a color directly, or to write a response as an ordered pair. If you are careful to write for the general case, you can change only the parameters at the top of the file and work with a new variation of Mastermind.

It is important to know, however, that you must start a new SML session when you change the variation of Mastermind. You cannot change the parameters "on the fly."

Using a Strategy Tree

We will get around to constructing an actual strategy tree in Problem 3. For now, assume that you have one and concentrate on writing functions to use it. We have provided strategies for several variations of Mastermind. They are named

`strategyCP.sml` and `naiveStrategyCP.sml`,

where `CP` stands for two integers, colors and pegs, indicating the variation of Mastermind. The "naive" strategies are the ones from Assignment 3. You must set the configuration parameters at the top of the file to match the strategy you are using.

Let us look at how the codebreaker uses the tree. If the codebreaker has the tree

```
Step ([Red,Red,Blue,Green], treeList)
```

then the codebreaker makes the specified guess, `[Red,Red,Blue,Green]`, and receives a response. That response corresponds to one of the trees in `treeList`. If the response is the, say, seventh in the list `allResponses`, then the seventh element of `treats` is the one that the codebreaker uses next. If that tree is `Win`, then the game is over. Otherwise, it will be another `Step`, giving the codebreaker the next move.

Notice that the last tree in the list corresponding to `winningResponse` is always `Win`.

1. [2 points] It is a little inconvenient to package the children of a node in the `knuth_tree` as a list, so we begin by isolating a tedious part of the process. Write a function `nextMove` that will, from a response and a strategy tree, select the strategy tree from the tree list that corresponds to that response. For example,

The function `funPairs` from earlier in the semester, or a variation on it, may be useful here.

```
nextMove (0,2) (Step([Red,Red,Blue,Green], treeList))
```

will return the element that is in the same position in `treeList` as `(0,2)` is in `allResponses`.

When a tree is `Win` or `Lose`, there is no next move, and the function `nextMove` ought not to be called. In those cases, resort to the convention of raising the `InternalInconsistency` exception (declared in the template file) to avoid warnings about incomplete matches.

```
nextMove : response -> knuth_tree -> knuth_tree
```

2. [2 points] The codemaker does not have a very interesting job. Once the secret code is chosen, the codemaker simply listens to guesses and gives responses. As reflected in our declaration, a `codemaker` is simply a function from codes to responses. The function

```
(matches [Yellow,Violet,Yellow,Orange])
```

is such a `codemaker`. Given such a function to impersonate the codemaker, we can simulate the codebreaker's use of the strategy. Write a function `play` that will, given a `codemaker` and a `knuth_tree`, produce the sequence of codes and responses leading to a win.

```
play : codemaker -> knuth_tree -> (code * response) list
```

The tree `Lose` should never be encountered; raise `InternalInconsistency` if it is. Adapting the example from Knuth's paper to our notation, the call

```
play (matches [Yellow,Violet,Yellow,Orange]) strategy64;
```

will produce the sequence

```
[ ([Red,Red,Orange,Orange],(1,0)),
  ([Red,Yellow,Green,Green],(0,1)),
  ([Yellow,Blue,Orange,Violet],(1,2)),
  ([Red,Green,Violet,Orange],(1,1)),
  ([Yellow,Violet,Yellow,Orange],(4,0)) ].
```

A smaller example comes from $(3,3)$-Mastermind. The call

```
play (matches [Yellow,Red,Orange]) strategy33;
```

produces the play

```
[ ([Red,Red,Orange],(2,0)),
  ([Red,Red,Yellow],(1,1)),
  ([Red,Yellow,Orange],(1,2)),
  ([Yellow,Red,Orange],(3,0)) ]
```

Constructing a Strategy Tree

We are now ready to build the strategy tree itself. Read Knuth's paper and take some time to think about the details before you begin to write code.

We will use the list of possible solutions to guide us as we construct the tree. The list will get smaller as we go deeper into the tree. What is the appropriate tree if the list of possible solutions is empty? What is the tree if the list of possible solutions has just one element?

When the list of possible solutions is larger, recall that we assign each guess a score:

- For each response to that guess, compute the list of remaining possibilities. (You did that on Assignment 3.)
- The score for that guess is the length of the longest list of remaining possibilities.

The strategy's guess is the one with the lowest score. There may be several lowest-score possibilities; break ties like this.

- Whenever possible, pick codes that are still in the list of candidate solutions. (We might get lucky and guess correctly!)
- When there is a choice, pick the one that is first in the candidate solution list.

See Knuth's paper for more details.

With the strategy's guess in hand, we can fill in the first component of a `Step`. Each response to that guess provides us with, as we saw above, a new list of possibilities. We can recursively generate a new tree from each such list, and those trees will be the elements of the list that is the second component of the `Step`.

3. [10 points] Write a recursive function `knuthStrategy` that will generate the strategy tree. Its only argument will be a list of possible solutions. The initial call will be with the list of all possible codes, and the recursive calls will be on shorter lists.

Be aware that a test code need not come from the list of candidate solutions. Knuth uses the example presented above to show that fact.

```
knuthStrategy : code list -> knuth_tree
```

Try your function on several different variations of Mastermind. Compare your results to the pre-computed strategies that we have provided. For the smallest variations, it is possible to compare strategies in precise detail.

Important: Do not include code to generate a tree in the file you submit. We will generate strategies of various sizes when we evaluate your code.

Hints and advice

- Make sure you completely understand what the tree should be and, algorithmically, how it should be created before you start coding.
- You will need to write several auxiliary functions. We are giving you a lot of flexibility in how you do this and it is critical that you think out how best to build up the final function out of smaller functions. An hour spent on a piece of paper or whiteboard strategizing will save you many coding headaches.
- Create your function incrementally. Write the auxiliary functions individually and test their functionality to make sure they work before putting them together.
- Work through a few examples by hand. Use examples from lecture or Knuth's paper to help you debug your functions. Start with small variations of Mastermind.

Exploring and Analyzing the Strategy

4. [2 points] Now we are going to write functions to evaluate the strategies that `knuthStrategy` creates. We want to know the maximum number and the expected number of rounds, over all the possible plays, for a given strategy. Write two functions that each take a strategy and return the desired number.

```
maximumRounds : knuth_strategy -> int
expectedRounds : knuth_strategy -> real
```

The idea is to create a list of all possible plays using the strategy—one for each possible solution. That can be done by mapping an appropriate function over `allCodes`. Then compute the number of rounds in each of those plays. From there, it is straightforward to find the maximum, or expected, value of the numbers in the list.

Notice that the expected number of rounds is a `real`. It is important to use the `real` operator `/`, not `div`.

See Table 1, in the appendix to this document, for the maximum and expected values for some of the variations of Mastermind.

5. [4 points] Up until now, the codemaker's role has not been very interesting. It is a simple task to compare a guess to the secret code and give a response. Resentful of the boredom, the codemaker decides to make the game more interesting. Instead of deciding on a secret code at the outset, the codemaker keeps its options open and give a response that *maximizes* the number of possible solutions remaining for the next move. The idea is to delay the success of the codebreaker. The codemaker does not want to reveal the dishonesty, so each response must be consistent with the previous rounds of the play.

Interesting question: Can the codebreaker, knowing the codemaker's strategy, ever force a win in fewer than five rounds?

Write a function `evilcodemaker` that implements our codemaker's behavior. Given a list of possible solutions so far and a candidate code, the function will return a response and a new list of possible solutions.

```
evilcodemaker : code list -> code -> response * code list
```

Break ties by returning the response that is earliest in the response list.

Another interesting question: Why do we have the evil codemaker computing the next move as the game is being played? Why not generate a data structure to reflect the evil codemaker's strategy, as we did with the `knuth_tree`?

The codemaker must expose the current list of possible solutions, but that poses no problem because the information is known by both players. Here is the beginning of a play of $(6, 4)$-Mastermind using `evilcodemaker`.

```
val candSolns0 = allCodes;
length candSolns0;
it = 1296 : int

val (resp1,candSolns1) = evilcodemaker candSolns0 [Red,Red,Orange,Orange];
val resp1 = (0,0) : response
val candSolns1 = [[Yellow,Yellow,Yellow,Yellow],...] : code list
length candSolns1;
it = 256 : int

val (resp2,candSolns2) = evilcodemaker candSolns1 [Red,Yellow,Green,Green];
val resp2 = (0,1) : response
val candSolns2 = [[Yellow,Blue,Yellow,Yellow],...] : code list
length candSolns2;
val it = 54 : int
```

I am so tired
the tree is not really for me
bradley and samuel

Sam Rubin '19, CS52 poet laureate

Appendix: Exploring Variations of Mastermind

Many human beings found the original six-color, four-peg game too easy. The toy manufacturer came out with a "deluxe" version which had five-peg codes. More recently, it produced an "ultimate" version with five-peg codes and eight colors. In addition, people have changed the rules to allow for some of the positions in the secret code to be empty; they effectively added a new color called `Blank`. You may be interested to experiment with your functions to see how the performance scales to five pegs or up to nine colors.

The Ultimate Mastermind game, mentioned in Assignment 3, is still in the lab. ... We hope.

Alternately, you may want to "scale down" for testing. In the template file, we have included code to adjust the number of colors and pegs. At the very top of the template file, you can set the numbers. There can be up to nine colors and an unlimited number of pegs. Finding the strategy for the standard $(6,4)$-Mastermind with six colors and four pegs is computationally intensive; it can take several minutes. We recommend that you use a smaller version as you develop your code. Then, when all is in order, you can generate and analyze the strategy for $(6,4)$-Mastermind.

Start small. There are only four codes in $(2,2)$-Mastermind, and you can see exactly what strategy you are producing. For comparison, we have given you the strategies for a few of the variations of Mastermind. Table 1 shows some of the results that you should expect as you experiment. You will probably be interested to complete the table.

		pegs				
		1	2	3	4	5
colors	2	2/1.500			4/2.750	
	3	3/2.000		4/2.740		
	4	4/2.500	4/2.812	4/3.375	4/3.590	5/3.991
	5	5/3.000		5/3.704		
	6	6/3.500	5/3.667		5/4.476	

Table 1: The maximum/expected number of moves using Knuth's strategy for variations of Mastermind.