Mastermind V1

Joshua Grizzell

CIS-7-48647
Fall 2021

Introduction

Title: Mastermind Version 1

Mastermind is a famous mind game that has a very simple concept: figure out the hidden code. The player gets ten turns to figure out the code. Each turn, the player guesses a code, and the game tells you information based on the guess. In my version of the game, if a number you guessed is present somewhere in the code, a "close" [~] box will appear. If a number you guessed is in the correct place somewhere in the code, a "correct" [!] box will appear. Otherwise, if neither of these conditions are true, an empty box [ ] will appear. The game requires the player to use **abductive reasoning** to figure out the code.

Example: The code is 2154. Keep in mind the player does not know this, and will need to figure out the code based on their guesses.

Guess	Result
`1111`	`[!][ ][ ][ ]`
`1234`	`[!][~][~][ ]`
`4152`	`[!][!][~][~]`

This version of mastermind is based on a very simple web game which also allows you to change the length of the code and allow duplicate numbers (or, in this case, colors) to be present in the code.

Summary and Discussion

Part 1: Making a Prototype

The first part of the project was, of course, to create a prototype based on the web game. I played the web game extensively, trying to figure out the ins and outs of the game, asking myself, "how might this have been implemented?" Once I felt I got a very good understanding of the game, I went to work coding it.

The web game uses 8 different colors that you slot into pegs to guess the code. Instead of colors, I opted to go for numeric digits, as it's essentially the same thing (besides, I don't know how to create interfaces yet.) I made a simple code generator function that returned an array of integers 1 through 8 (simply named `code` in the program.) I debated whether to use an enumerator with the different colors to replicate the web game, but that just seemed redundant and rather annoying, so I left it as basic numeric digits.

After the code generator was created, I ran into another problem: input verification. I had to make sure that users couldn't input digits 9 or 0, because those weren't in the game. So I made a function `chkDgts` that returned a boolean value true or false based on whether or not the digits in the user's guess were all legal.

Now that we've finished the input verification, we can move on to the actual "game" part of the program. After it's been verified that the user's input was valid, we then call the function `cnv2Arr` which takes the user's input and converts it into an integer array. This integer array can then be compared with the secret code generated at the beginning of the program. To display the hints after each guess, I created a function `cmpArr` which returns a concatenated string. I used a concatenated string to add the [ ], [~], and/or [!] boxes (as described at the beginning of this paper) based on the results from comparing the two integer arrays.

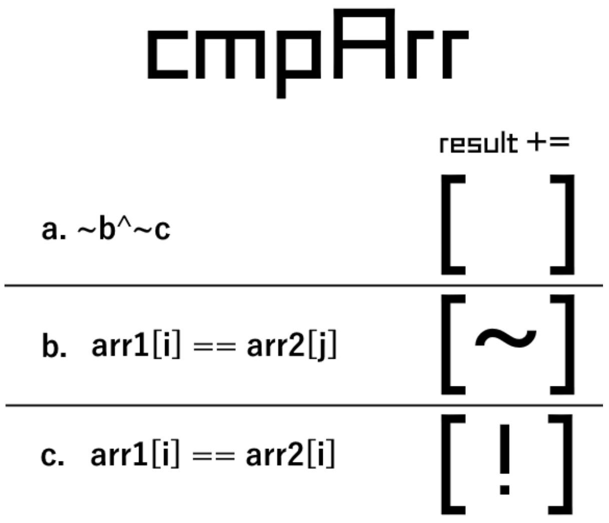
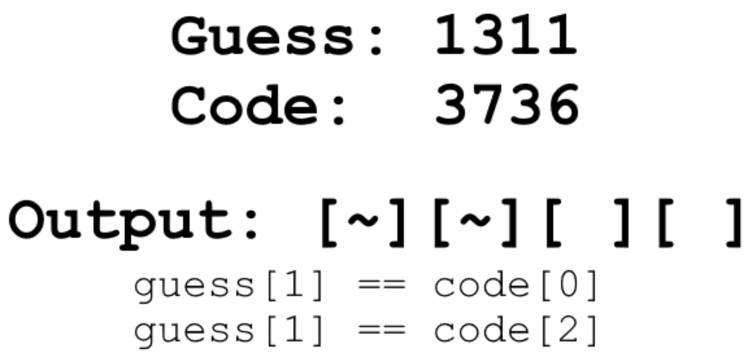


Figure 1.1: Illustration showing how the concatenated string "result" was determined in function cmpArr.

After the string is returned from the `cmpArr` function, it is displayed in the console to give the user their hint. The process continues until the game is finished. The prototype has been completed!

Part 2: Fixing Bugs

The game works, but it's just a prototype. There's still many things to add, and I already noticed a major bug. When duplicate digits are present in the code, even if there is only one of that digit in the guess, the `cmpArr` function will flag all numbers in the guess as "close." This is kind of hard to explain, so I created a figure to demonstrate the bug.

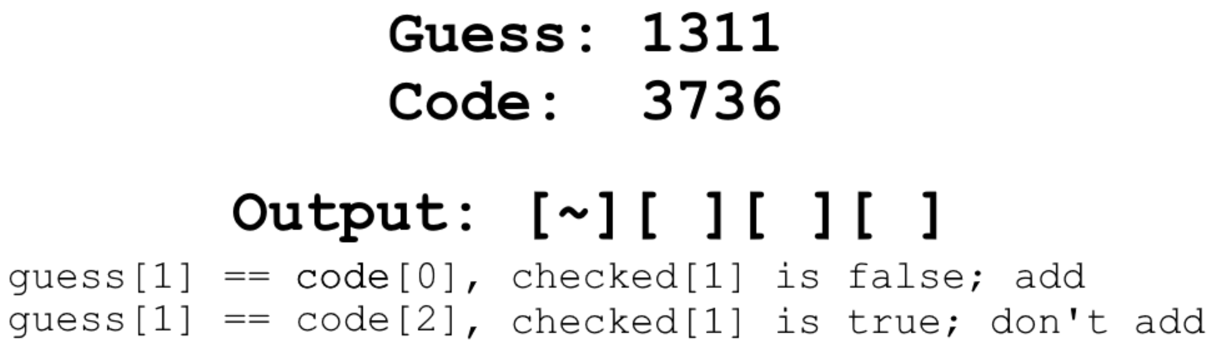


```
Guess: 1311
Code:  3736

Output: [~][~][ ][ ]
guess[1] == code[0]
guess[1] == code[2]
```

Figure 2.1: A bug created by flagging a single digit as "close" to two identical digits. That still doesn't sound right… eh, just look at the picture.

To fix this, I had to add a boolean array to the `cmpArr` function. This boolean array, called `checked`, would initialize as false. Once a digit in the `guess` array has been flagged as "close", the corresponding boolean array will be set to true. The digit has been checked. If the program detects that the specific digit in the guess array is "close" to any more digits in the code array, it will notice that the checked array for that digit has already been flagged as true, not adding another "close" box to the concatenated string.



```
Guess: 1311
Code:  3736

Output: [~][ ][ ][ ]
guess[1] == code[0], checked[1] is false; add
guess[1] == code[2], checked[1] is true; don't add
```

Figure 2.2: A solution to the bug outlined in Figure 2.1

Of course, on top of this bug, I had to fix a few others (`chkDgt` not working properly because it didn't detect the first digit in a guess, among other things,) but the `cmpArr` function was the most difficult to figure out how to deal with. With the game's bugs ironed out, it's time to polish.

<u>Part 3: QoL Improvements</u>

I quickly realized even during the prototype phase that the player actually needs to see their previous guesses and results to actually try to reason out the code. To do this, I created two arrays, an integer array and a string array, and a function, `prnHist`. Instead of just displaying the concatenated string outright, I would store it in an array, along with the guess from the beginning of the round, and put both arrays into `prnHist`, which I'd run at the end of the round. This function would print the history of the game; the player's guesses and the result of those guesses, including all prior rounds. I also included a proper winning mechanic for the game, albeit a little hacky.

Before each game began, I also implemented two short explanatory sentences on what exactly to do in the game. That's pretty much it, though.

Part 4: So Many Options

Up to this point, I didn't have an option to turn off duplicates, and the code length was hard set to 4. It was finally time to implement some of the options that the web game offered to the player.

The entire game was played in main, which I swiftly changed. I pasted the game into a new function called `mstrmnd`, adding two function parameters to it as well called `codeLen` and `dupes`. `codeLen`, as you can probably tell, is the length of the code as an integer. `dupes` is a boolean that, if true, allows duplicates to be present in the code. Otherwise, the function `genCode` will go out of its way to ensure there are no duplicate digits in the code.

Both of these function parameters are selected at the beginning of the program by the user using switch case statements.

The game is complete!

Pseudocode

(since the main function is super simple, I included the mstrmnd and cmpArr functions as well.)

```
Initialize program

main
    Ask the user to select a length of the code.
        Code length can be 4, 6, or 8.
    Ask the user if they want duplicate digits or not.
        Can be true or false.
    Begin the mastermind game
    Exit program (0)

mstrmnd
    Initialize arrays “code” and “urCode” with the length given earlier by the user.
    Generate the secret code (genCode)
    Explain rules of the game concisely
    For loop (10 times):
        While the number is not valid:
            Have user input their guess
            Check digits to ensure number is valid
            If valid:
                Put guess into history array
                Convert the guess to an integer array
        Compare the user’s integer array with the secret code array (cmpArr)
        Store the result from the comparison in a string array.
        Print the history of the guesses and results
        If the player solves the code:
            Print a basic you win message or something
            Return
    Print a basic you lose message with the code
    Return

cmpArr
    Initialize boolean array “checked” as all false
    Initialize integer variables “correct,” “close,” and “total”
    Initialize string “result” as an empty string

    For loop (i < code length):
        If arr1[i] is equal to arr2[i]:
            Add 1 to correct
            Continue
        For loop (j < code length):
            If arr1[i] is equal to arr2[j]:
                If the digit has not already been checked:
                    Add 1 to close
                    Set checked[j] to true
                Break
    If correct is equal to code length
        Add “you win” to the result string
        Return result
    While correct is being decremented by 1:
        Add “[!]” to the result string
        Subtract total by 1
    While close is being decremented by 1:
        Add “[~]” to the result string
        Subtract total by 1
    While total is being decremented by 1:
        Add “[ ]” to the result string
    Return result
```