Homework 2: Mastermind

Suggested Work Timeline (attempt to complete the following sub-tasks):

- Randomly generate a secret code
- Compute how many letters are in the correct position
- Compute how many letters are out of place
- Create the interactive game script

Collaboration Policy: We highly recommend that you complete this assignment in **groups of two**.
You may discuss high-level ideas with other groups, but you may not share your code with anyone but your partner. Please refer to the course syllabus for clarification on academic honesty policies.
If you are working with a partner, only one group member needs to submit the assignment.
If you want to work alone, submit your assignment to Eduko as you did for HW1.

Starting Materials

You should start from your homework 1 code. You are welcome to save the following template to your computer and "**Import …**" it into your homework 1 Snap! file. This will give you the skeleton version of the blocks described below, under "Variables". **Note that you are required to build the blocks described below, following any restrictions indicated. Therefore, please read the requirements for each block.**

Description

The goal of this homework assignment is to create the full infrastructure for the game Mastermind (a code guessing game: https://en.wikipedia.org/wiki/Mastermind_(board_game)). The "flow" of your program is that the computer generates a secret code and the human player tries to guess the secret code, using the feedback she received from previous guesses.
A secret code is a string of letters (no spaces), where each letter is one of ROYGBV (the first letters of the standard colors: Red, Orange, Yellow, Green, Blue, Violet). You are allowed to have repeated letters in your code. Example secret codes: G, GR, GRO, GYY, RRBG, OYOO, BROY, VVVVV, etc.

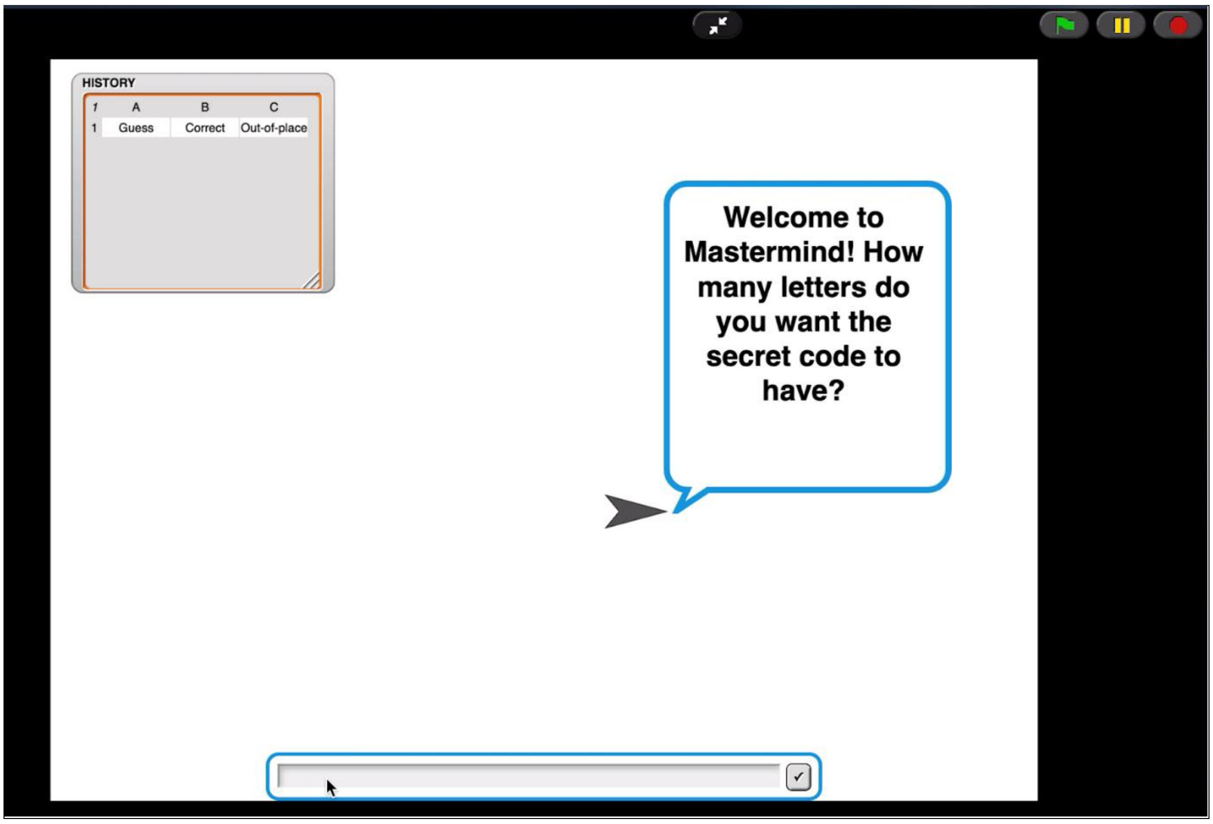
The game begins with the computer asking how many letters the secret code should have. The computer then generates a random code of this length. The computer should announce the number of letters in the secret code, but not the code itself. The computer then asks the player to try to guess the code. Both the secret code and the guess should be case-insensitive (i.e. rrrr = RRRR).

After the player makes a guess, the computer should tell the player whether the guess was right or wrong. If the guess was wrong, the computer should also tell the player **how many letters in the guess were correctly positioned in the code** and **how many were in the code but out of place**. (This is different than what you did in HW1, but you may use HW1 as a starting point). If the guess was invalid (too short, too long or invalid characters), the computer should display an appropriate error message, as described in the requirements section.

When the user guesses the correct code, the computer should report how many valid guesses it took the player to get the correct code. Remember to count only the valid guesses (i.e. a guess with too many letters, or a guess with letters besides those in ROYGBV in it would be an invalid guess)!

Sample Game Video

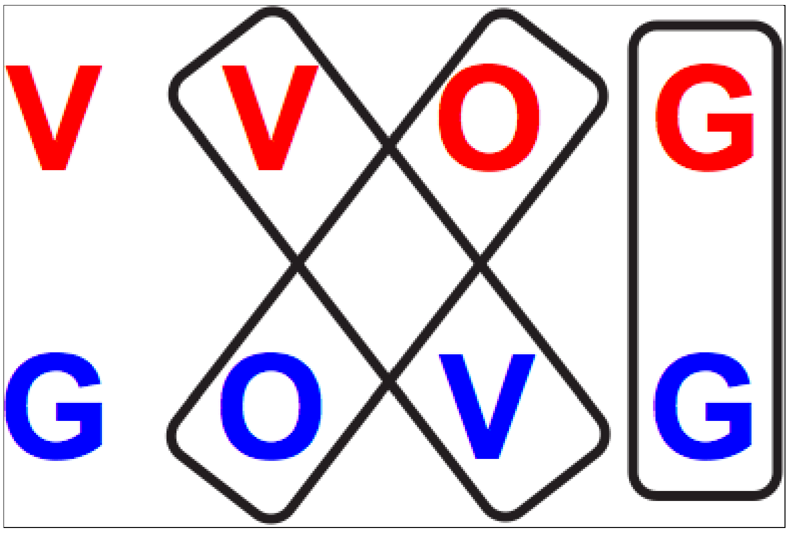
Here is an example of the game play to give you more of an idea how this works. Scroll to the end of this spec to read the following video's transcript, which includes an optional commentary on how the player thinks.



Welcome to Mastermind! How many letters do you want the secret code to have?

Algorithm for Required Blocks

Let's make sure you understand how to correctly count letters of a guess as "correctly positioned" and "out-of-place". Essentially, it's all about pairs. You first find all the pairs that line up across from each other (i.e., it's the same letter in the same position in the code and in the guess). Circle these and count them as "correctly positioned". The reason to circle them is so that they can't be chosen again as "out-of-place". The number of pairs circled is the number of letters you tell the user are "correctly positioned". Now, for computing how many letters are "out-of-place", find an uncircled letter in the code that appears in the guess and is also uncircled in the guess. That's an "out-of-order" pair. Circle that pair (so it can't be chosen again) and keep going. When you're done with all these pairs, that's your number of "out-of-place" letters.



Here's an example. The computer chooses the code VVOG. The user guesses GOVG. How many are "correctly positioned" and how many are "out-of-place"? Using the algorithm above, the rightmost Gs is the only pair across from each other, so they get circled, and this means there is **one letter in the guess that is "correctly positioned"**. Then for all of the other letters, we look for uncircled pairs. We circle the Os, and the lower V with either one of the upper Vs *(but not both)*. Since we had 2 pairs (O and V), that's **two letters in the guess that are "out-of-place"**. So we tell the user that for their guess GOVG, one letter is "correctly positioned" (G) and two are "out-of-place" (O and V).

Oh gosh (you say), how do you do circling in Snap*!*? You mean use the **`move`** and **`turn`** blocks inside a **`repeat`**? No. We don't need to actually circle letters (though it is a useful way of understanding this algorithm visually and conceptually). Rather, by "circling letters" we mean that we want to remove those letters from being considered after they've already been counted, either as "correctly positioned" or "out-of-place." We don't want to double-count any letters.

Here's a really cool way to count "out-of-place" letters -- after you've removed the "correctly positioned" pairs, you keep track of how many of each letter (ROYGBV) remains (uncircled) in the guess and in the code, like a histogram. Then you match up the counts for each letter and take the minimum--that represents how many copies of each letter are "out-of-place". Sum those up and you have how many letters are out of place in total. Here's an example (**hint**: this table is a really useful way to approach this problem):

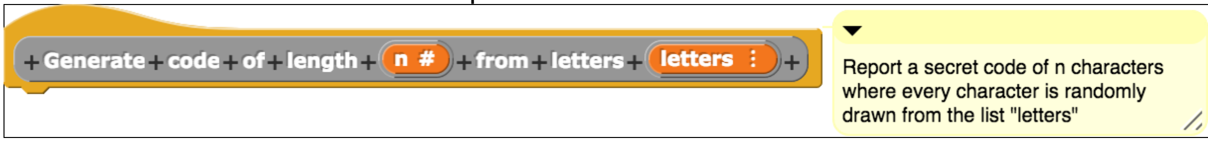
Initial	After correct pairs removed	R	O	Y	G	B	V	Sum of the `mins` (# of out-of-order letters GOVG)
VVOG	VVO	0	1	0	0	0	2	
GOVG	GOV	0	1	0	1	0	1	
	`min` for each letter	**0**	**1**	**0**	**0**	**0**	**1**	**2**

Requirements

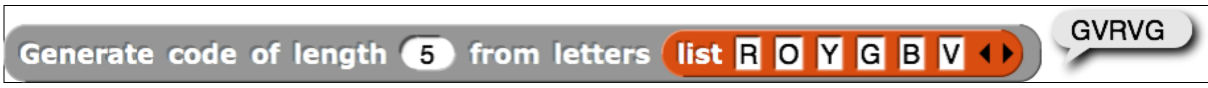
You are required to build the following blocks. Solutions that do not build and use these blocks will be penalized. Also, please note that each of these reporters has

inputs. You should be giving your blocks inputs as demonstrated. Global variables should not appear inside any of the blocks shown below.

1. Randomly generate a secret code of a given length. This is done by authoring this reporter block:

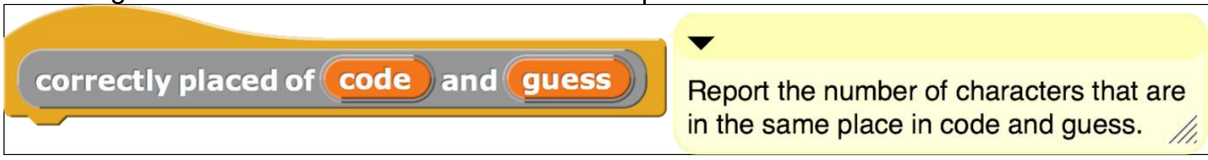


Report a secret code of n characters where every character is randomly drawn from the list "letters"

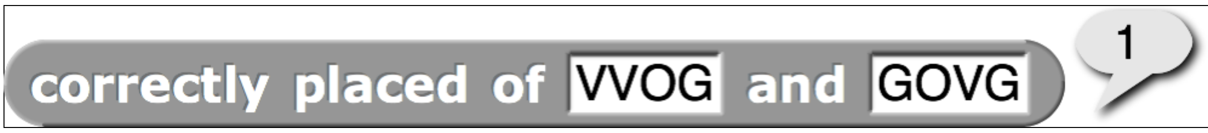


---

2. Compute how many letters are in the correct positions. This is done by authoring this reporter block:



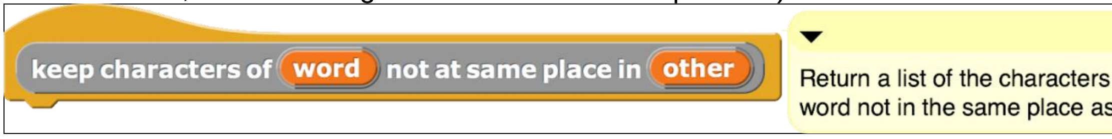
Report the number of characters that are in the same place in code and guess.



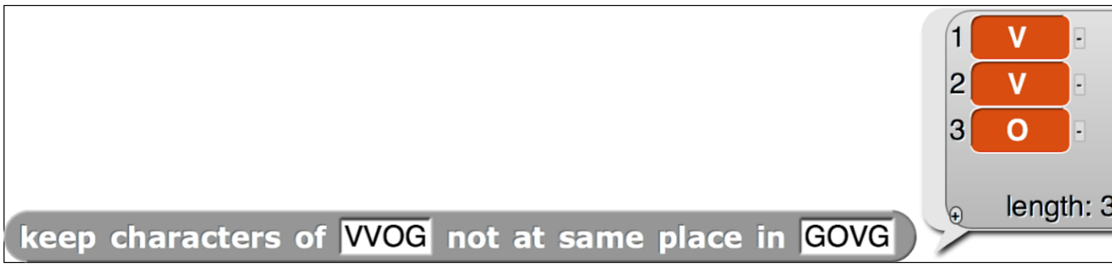
---

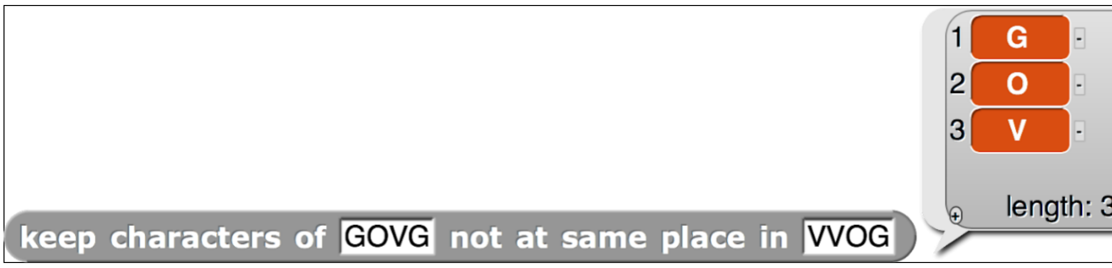
3. Compute how many letters are out of place. We'll help you with this block, since it's complicated. Parts (a) and (b) are helper blocks that you will want to utilize in part (c).

   1. Report a list of the letters of a first word that are not in the same position as the other word (this will have the effect of "circling" the correct letters, and removing them from further computation)



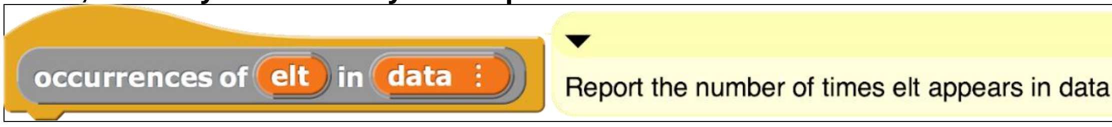
Return a list of the characters word not in the same place as

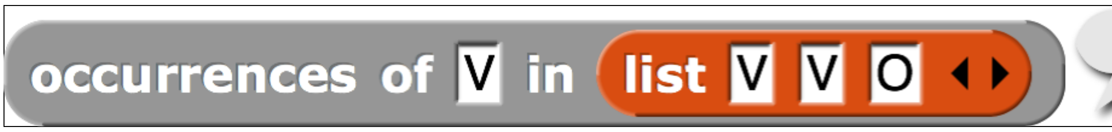




---

2. Report the number of times an element appears in a list (this will help compute your histogram). **You are required to use HOFs for this block, and may NOT use any for-loops.**

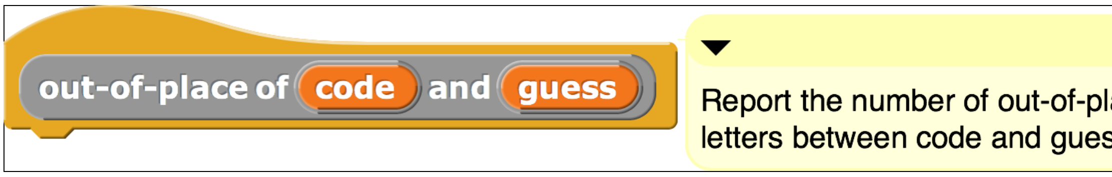

occurrences of elt in data
Report the number of times elt appears in data

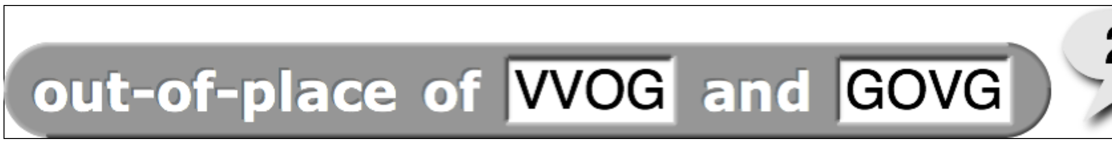


---

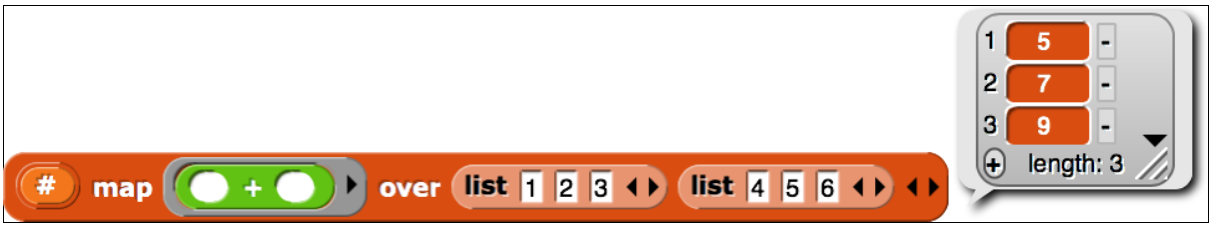
3. Finally, write the block that counts the letters “out-of-place” between the code and the guess. This block should utilize the helper blocks. **You are required to use HOFs for this block, and may NOT use any for-loops.** In particular, use the `map` block shown below, which can be imported by going to Libraries -> List utilities. Notice that this `map` allows you to map over two lists in parallel, as shown in the example below. You can ignore the # variable. You might also find a `min` block useful -- similar to `max` from lab.

   If you’re unsure where to begin, read the algorithm described below which explains how to implement this block.

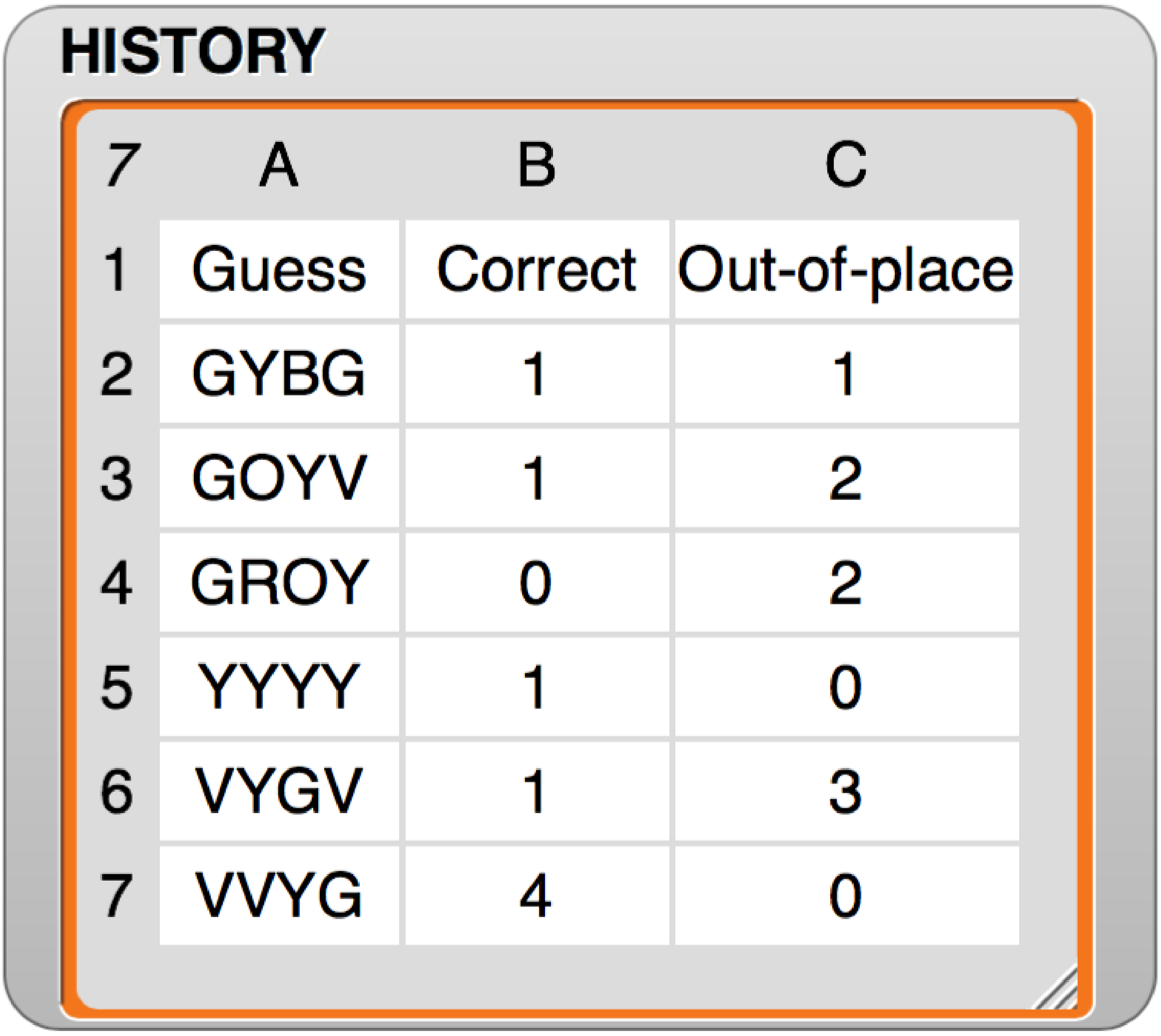

out-of-place of code and guess
Report the number of out-of-pl
letters between code and gues



Here is an example of how to use map with two lists.



4. Keep a running history of the user's guesses and the number of correctly positioned and out-of-place. This means declaring a variable HISTORY that is a list-of-lists that the user can refer to when making a new guess. That is, it's a list of all the rows, where each row is a list of three elements: {the "Guess", the number of characters in the "Correct" position, and the number of characters "Out-of-place"}). The first row is the column headers. For example, this would have been the history at the end of the example game earlier:



HISTORY

7	A	B	C
1	Guess	Correct	Out-of-place
2	GYBG	1	1
3	GOYV	1	2
4	GROY	0	2
5	YYYY	1	0
6	VYGV	1	3
7	VVYG	4	0

5. Make sure all user inputs are valid and tell the user when it is not and why it is not. These come in three forms:

1. There are too few characters
2. There are too many characters
3. The characters are anything but “ROYGBV”.

Here is a sample interaction that shows the valid guess check. In this example, the secret code has 4 letters. Your code does not need to match this exactly, but it should account for all three conditions of invalid inputs.

Video *Transcript*

Player: ROY

Computer: Guess too short; it has to be exactly 4 letters. Please guess again.

Player: ROYGBV

Computer: Guess too long; it has to be exactly 4 letters. Please guess again.

Player: BJC!

Computer: Guess can only contain the letters ROYGBV. Please guess again.

---

6. Your game should have at least one sprite, but there are no graphical requirements other than that. Feel free to develop the graphics more if you'd like -- the practice will come in handy during later projects!

Reference Table

You are **required** to build the following **5 blocks** (as described above).

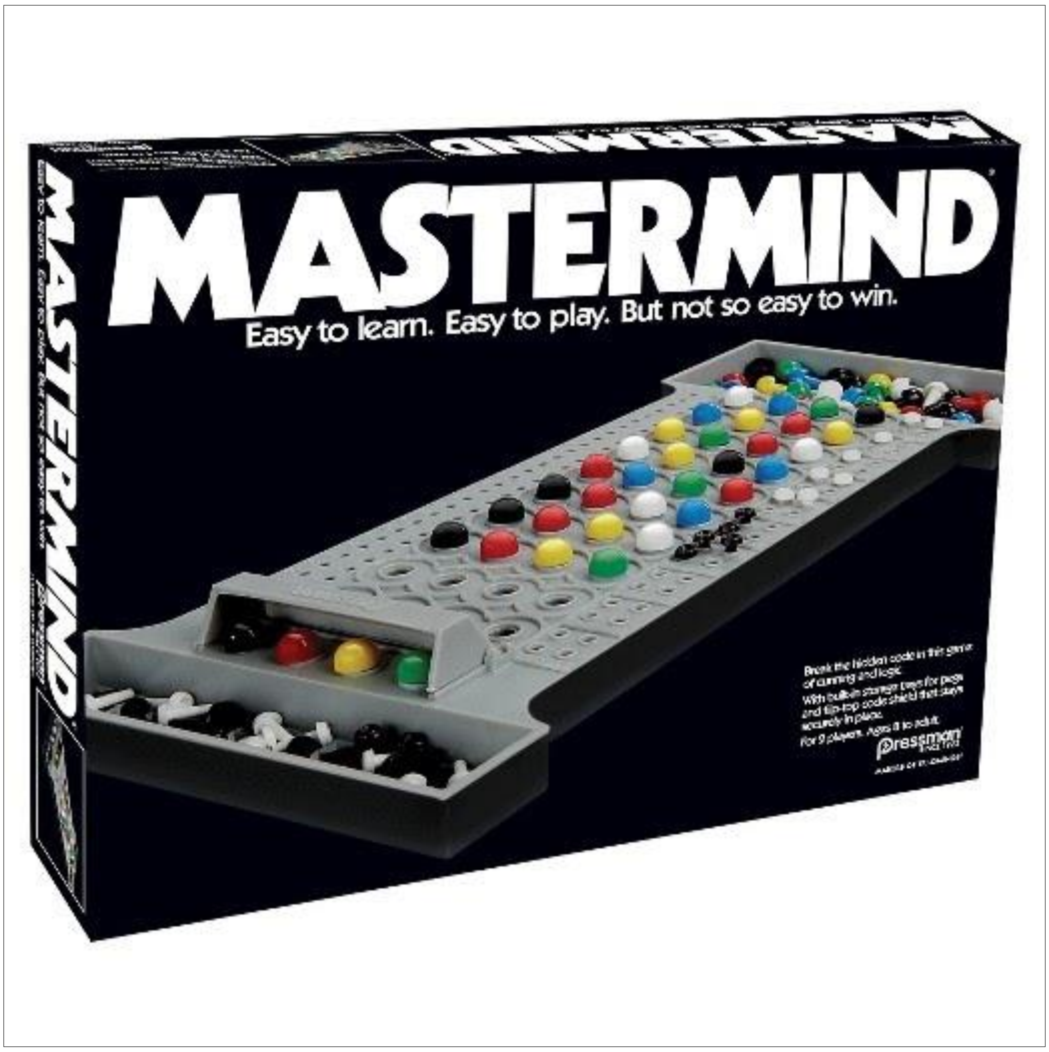
Block	Input(s)	Output	Restrictions
generate code (Requirement 1)	**n:** number of letters in the secret code **letters**: list of letters (R, O, Y, G, B, V)	A secret code where every letter is randomly drawn from the “letters” list	No global variables.
correctly placed (Requirement 2)	**code:** a word representing the secret code **guess:** a word representing the user’s guess	The number of letters that are in the same place in “code” and “guess”	No global variables.
keep characters not at same place (Requirement 3a)	**word:** first word **other:** second word	A list of the characters in “word” not in the same place as “other”	No global variables.
occurrences (Requirement 3b)	**elt:** a character **data:** a list	The number of times “elt” appears in “data” list	No global variables. HOFs only. No loops.
out-of-place (Requirement 3c)	**code:** a word representing the secret code	The number of out-of-place letters between “code” and “guess”	No global variables. HOFs only. No loops.

	guess: a word representing the user's guess		

Extra for Experience

There are two ideas, and you can do the first or the second, or both:

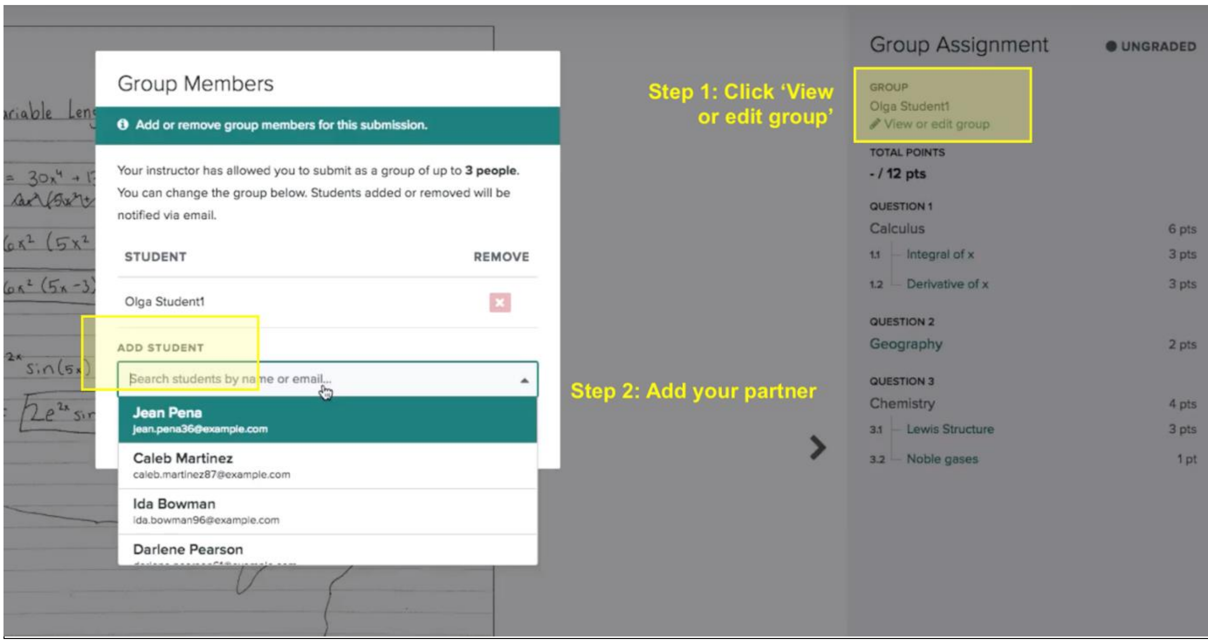
1. **Implement a full Graphical User Interface (GUI).** Game play would not involve an **`ask`** block for input (the user would instead click or drag the virtual pieces to the slots to make a guess) or a **`say`** block for output (the user would see up to four small black and white pegs for output, the traditional colored coding is black for "correctly positioned" and white for "out-of-place"). Note that it's traditional to have the four output pegs NOT shown left-to-right to imply there's some connection between the location of the output pegs and the guess – they typically put the output pegs in a 2x2 grid as shown below. Here's the box from Target to give you an idea how people actually play this game with pegs (note their colors aren't ROYGBV).



2. **Flip the human-computer script!** Instead of the computer generating the code and the human guessing, have the human put in the code and make it the computer's job to guess it. (The computer would obviously also do the gruntwork to calculate the "correctly positioned" and "out-of-order" responses, since you'd already have that machinery in place). The important thing to note is that every computer guess has to be consistent with all the previous "correctly positioned" and "out-of-order" responses -- that is, every guess could be the solution. (That wasn't the case with our guess of "**YYYY**" in the sample game, but was the case of all our other guesses) Keeping consistent is actually pretty easy, you could start with a list of all the possible codes, and keep filtering it with every guess once you see the "correct" and "out-of-order" responses (hmm, I wonder what block could help with that), and picking one randomly from that list. There are obviously ways to be more optimal about it than by picking randomly, so if you come up with some clever way to choose the "best" move, make sure you indicate that in your writeup. Since the flow of this game is so different than the original, and we'd like to keep it to one submitted file, you should have a "do you want to play the original or the flip-the-script game" (or something like that) that will let the user choose between the human-guesses game and the computer-guesses game.

Submission Guidelines

1. **(required) project file** - Export your Snap! project. Name this file *HW2_FirstnameLastname.xml*
   Upload your xml file to Eduko under the HW2 assignment.



Group Members

Add or remove group members for this submission.

Your instructor has allowed you to submit as a group of up to **3 people**. You can change the group below. Students added or removed will be notified via email.

STUDENT	REMOVE
Olga Student1	✕

ADD STUDENT

Search students by name or email...

Jean Pena
jean.pena36@example.com

Caleb Martinez
caleb.martinez87@example.com

Ida Bowman
ida.bowman96@example.com

Darlene Pearson

Step 1: Click 'View or edit group'

Step 2: Add your partner

Group Assignment ● UNGRADED

GROUP
Olga Student1
View or edit group

TOTAL POINTS
- / 12 pts

QUESTION 1		
	Calculus	6 pts
1.1	Integral of x	3 pts
1.2	Derivative of x	3 pts
QUESTION 2		
	Geography	2 pts
QUESTION 3		
	Chemistry	4 pts
3.1	Lewis Structure	3 pts
3.2	Noble gases	1 pt

2. **(optional) extra credit sprite** - Make a new sprite by right clicking on the original *Sprite* in the Sprite Control Area, and selecting "Duplicate". A new sprite appears as *Sprite(2).* Rename *Sprite(2)* just above the scripting area to *Extra Credit.* Now, right click anywhere inside the Scripting Area and select "add comment". Edit the default text to briefly explain how you have completed the extra credit.

Tips

- ABSTRACT! For every sub-task, sub-sub-task, etc., **make a block**! Once you write a block, you can use it in your code wherever you'd like and the details of its implementation get abstracted away. This makes your code *easier to read* and *easier to debug*. Plan/design ahead: think about what blocks you'll want to have. Always break down larger problems into smaller sub-problems.
- Don't hard-code anything! Take advantage of loops (eg. for i, for item, repeat) and HOFs instead of having long scripts with many if-statements. Generalize!
- Save your code often and back it up using a free cloud storage service like DropBox or Google Drive.
- Code a little bit at a time and test as you go.
- In a perfect world, users would read all of the user documentation provided and interact with a program exactly as the designer intended. But as we all know few people in the real world actually read documentation. We have to design good programs to prevent user mistakes or mitigate them. Most importantly, we have to make sure user error does not break our code. Remember to handle the situation when the user guesses a word that is too long or too short, for example.

Sample Game Video Transcript with Commentary

Here is the video transcript of the sample game play included in this spec. *The words in [square brackets] are our commentary. Feel free to ignore the green "player thinks" commentary, we included it to give an example of how to play the game well.*

Video Transcript

Computer: Welcome to Mastermind! How many letters do you want the secret code to have?
Player: 4
[the Computer has just generated the secret code ***VVYG****, which has 4 letters.]*
Computer: There are 4 letters in the secret code. What's your guess?
Player: GYBG *[first guess]*
Computer: No, but 1 is/are "correctly positioned" and 1 is/are "out-of-place".
[G is "correctly positioned", Y is "out-of-place". We don't count G twice since there's only one G in the code.]
[player thinks, maybe the first G was right, but the Y was shifted over, and there is no B and no second G, so let's ask about some of the letters we haven't tried yet, O and V]
Player: **GOYV** *[second guess]*
Computer: No, but 1 is/are "correctly positioned" and 2 is/are "out-of-place".
[Y is "correctly positioned", V and G are "out-of-place". We don't count V twice from the code since there's only one V in the guess.]
[player thinks, maybe the first G has been in the right place all along, the O and Y need to be shifted over, and we haven't tried R yet]
Player: GROY *[third guess]*
Computer: No, but 0 is/are "correctly positioned" and 2 is/are "out-of-place". *[G and Y are "out-of-place"]*
[player thinks, "I wonder how many Ys there are?"]
Player: YYYY *[fourth guess]*
Computer: No, but 1 is/are "correctly positioned" and 0 is/are "out-of-place".
[player thinks, there's only 1 Y, only 1 of {G,B,G} (from the first guess), only 2 of {G,O,V} (from the second guess) and only 1 of {G,R,O} (from the third guess). They wonder, hmm aside from a single Y, what if there's no G. Then there has to be a B (from the first guess), there has to be both O and V (from the second guess) and the third guess is

consistent and doesn't tell us anything (the O counts for the 1). So our guess will be some combination of YBOV, but in what order? From the first guess we know that 1 of is "correctly positioned" and 1 is "out-of-place" from -YB-, from the second guess we know 1 is correct and 2 are "out-of-place" from -OYV and from the last guess that both are out of place from --OY. So from the first guess let's say Y was right and B is in the front (i.e., BY--), then from the third guess O would have to be in the end and therefore V in the 3rd spot as BYVO, but that wouldn't have 1 correct from the second guess, so that can't be right. Hmm. Ok, what if Y was still right but B was at the end (it couldn't be in the 3rd spot from the first guess). So that's -Y-B, but the second guess tells us that 1 has to be correct from -OYV and that's impossible since the 2nd and 4th spot are wrong and Y has already been claimed so the 3rd spot is wrong, so therefore (backing up) Y can't be right from the first guess. Therefore B is correct in the 3rd spot and Y can't be in spot 2 (or spot 4 from the third guess, which means it has to be in spot 1. So that's Y-B- but the remaining spots are O and V and if it's YVBO then the second guess won't have ANY correct (it was supposed to have exactly 1), and if it is YOBV then it will have 2 correct. So that can't be right, and (backing up even further), we know it can't be YBOV. So then our assumption of "what if there's no G" is wrong, and (aside from the single Y), there HAS to be a G. Then from the first guess there's no B and not two Gs. From the third guess there's no R or O. If there's no O, and the second guess has to have 2 of {G,O,V} then there's a V. So now we're narrowing it down. For sure, there's YGV and not BRO (and no second G). So either Y or V is duplicated. Let's guess and say it's a duplicated V (if we're wrong then it has to be a duplicated Y). So we're down to YGVV, but in what order? Let's now make a table of what we are sure so far.

code=VVYG Guess	**Correctly Positioned**	**Out-of-place**	**Showing only YGVV letters**
GYBG	*1*	*1*	*GY-G (but not sure which G)*
GOYV	*1*	*2*	*G-YV*
GROY	*0*	*2*	*G--Y*
YYYY	*1*	*0*	*YYYY (but not sure which Y)*

...player still thinks. Look, there are two Vs, so there a higher chance that the second guess's last V is right. So let's say that V is last at ---V. Then Y can't be in 3rd spot and G can't be in the 1st spot (which we already knew for sure because of the third guess). Then Y has to be in the 2nd spot from the first guess so that's VYGV (since G can't be in the 1st spot so it must be in the 3rd and therefore the second V is at the front). We're all consistent, let's go for it.]

Player: **VYGV** *[fifth guess]*

Computer: No, but 1 is/are "correctly positioned" and 3 is/are "out-of-place". *[The first V is correctly positioned]*

code=VVYG Guess	**Correctly positioned**	**Out-of-place**	**Showing only VVYG letters**
GYBG	*1*	*1*	*GY-G (but not sure which G)*
GOYV	*1*	*2*	*G-YV*
GROY	*0*	*2*	*G—Y*
YYYY	*1*	*0*	*YYYY (but not sure which Y)*
VYGV	*1*	*3*	*VYGV*

[player thinks, where was my last assumption to back up. I assumed V was at the end (the positions of all the other pieces followed logically), and we didn't get it, therefore V is not at the end. From the second guess, we now know that Y is in the 3rd spot (remember G can't be in the 1st spot, and somebody has to be correct in that guess). So from the first guess G has to be at the end (since somebody has to be correct, and it's not Y or the first G). So we know it's --YG and there's two Vs left for the first two spots so it HAS to be VVYG. We got it, and we don't even need the computer to tell us!!!]
Player: **VVYG** *[sixth guess]*
Computer: Yes, that is correct! You figured out the secret word in 6 guess(es).
[...and the game ends]