Story 1:

I had a lot of fun during my work for the capstone project for the foundation portion of DevMountain coding boot camp. I wanted to do something which is a little different than the things I did up to that point in this course. Up to that point I mainly worked with static elements of web pages, some of these webpages were connected to a back end server, and many had a javascript file embedded within the HTML file for adding functionality to the static elements. I also separately worked on more complex javascript codes which involved conditionals and loops, but they were not connected to any HTML web pages. So for my capstone project, I wanted to add more complex javascript logic to my HTML elements, and have them animate, meaning move around in specific ways under specific conditions. For the capstone project, I learned a very cool functionality of javascript for applying animation to web pages, which uses a recursive loop with a controllable time delay between each iteration of the loop. I learned many other things for this project, some of them were related to CSS, which I was a little weak on up to that point. I learned to retrieve and control CSS properties of HTML elements in an object oriented manner, by which I mean I created classes of HTML elements and created objects from those classes for assigning to those objects methods and attributes. Essentially the end result of the project was a game of soccer ball shooting, where user of the website can control speed and angle of the ball, and can shoot the ball to a goal which is changing location randomly with each score, and avoiding a moving goalkeeper which moves faster and faster as score increases, but reaches a terminal velocity for a specific value of the score. Score incremented by one point or four points, depending how the ball was shot toward the goal. I stored, retrieved, and updated key variables in the back end and communicated with the back end using axios. Many of the functionalities, as I was adding them, proved more challenging than I thought, but also fun. As website features started to pile up, debugging became increasingly difficult, because every new feature had dependencies with all the existing features, and if something went wrong, it was hard to pinpoint it. I paid close attention to and optimally configured all the small and sometimes overlooked aspects of all the features, logic, and functionalities, covering all the edge cases, so that no matter what the user did, the game would progress in a satisfactory and consistent manner.

Story 2:

During the foundation portion of the DevMountain coding bootcamp, me and the other students were given a different coding problem to solve during every morning. I really had a lot of fun with those coding challenges, especially the last one, which was a game theory problem and it asked me to code a program which can be used by a player to determine if he can win a game of takeaway. Per the rule of the game, the player can take 2, 3, or 5 stones from a collection of stones at the first turn, and take turns with another player, who also has to follow the same rule. If a player cannot take any stones at any point of the game, he will lose the game. I used a "for" loop with three iterations, combined with a recursion process, to write a function which essentially simulated the entire game, and checking every possible permutation of moves by the first player and counter moves by the opposing player to check if there is a guaranteed path to victory for the first player. The recursion aspect of the function allowed it to call itself, and each

time it would do so, that would simulate a single turn in the game, as part of the sequence of turns from the first turn to the last turn. Essentially, each recursion will call a "for" loop, and each iteration of the "for" loop will call a recursion. The FILO (First In Last Out) nature of the recursion stacks meant that when the recursion reached the last stack, meaning when the number of stones remaining is 0 or 1, it would start sending a "win" or "lose" signal for the first player, back up the recursion chain. The "for" loop allowed each of the options for the stones (2, 3, or 5) to have its own specific recursion branch. The "for" loop can be broken if any turn belongs to the first player and signal coming from the bottom is a "win", since we know that the first player is in control of that turn and just needs to have one path to victory, as we only care about him winning. The "for" loop can also be broken if any turn belongs to the opposing player and signal coming from the bottom is a "lose", since we know that if the opposing player is in control of the turn, every path possible will have to lead to the first players victory, so that the opposing player has no way of changing the outcome. In order to assist me in solving this problem, I made recursion branch diagrams on a 11" by 17" paper, which helped me visualize the flow of the program. When I ran the program, it followed a consistent pattern and accurately predicted if the player would win or lose, given the number of stones available at the starting turn.