This project was much nicer than the last one. I followed through the instructions step-by-step. Most of everything was straightforward, things I could figure out easily. The hardest part - obviously - was the recursive solve and solve_shortest methods. Another thing that tripped me up at first was the way we were storing the linked list stack. I started out by adding everything to the end of the linked list and popping from the end of the list. This does work, but takes too much time and could be more efficient. After switching to the more efficient method I passed all the tests and everything worked great.

To do this more efficient method, instead of looping to the end of the linked list, simply create a new node with the next node being the current head. Store the new node as the new head. When popping you simply store the data from the head, and set the head to head.next. This cuts out any loops from the equation, and makes pushing and popping take the same amount of time no matter how long the stack is while still following FIFO rules.

To solve the solve method, I started out by realizing that for every square that I'm at, I should check what happens if I go up, down, left, and right. There are multiple base conditions: If we are at the end, return the current path. If start is in the explored list, meaning we are backtracking, return. If we are out of bounds, return. If we are in an ocean, return. After looking through all possibilities from the point we are now, if none of them work out remove the current spot from the current_path and explored list. If there are no paths, then all 4 recursive methods will return None. If any of the recursive paths return a path, we will return that first path.

For solve_shortest I chose to store the current path as a list, and use the helper method to convert it to a LLStack and store that as the best_path. If we find a path we convert it to a list. One of the base conditions is that if the length of the current path is greater than the size of the best_path, return. No point in looking for a path that is longer than what we've found. After checking every possible path - and stopping if the path is longer than one we've already found - then we can return the best_path we found.

My test cases are enough because they cover every major portion of each class. I cover each variable/property, and every function. I test these in a variety of ways, including incorrect types, incorrect values, etc. I input these for the property as well as the init function to make sure the starting checks happen for both cases. I make sure that they are storing the LLStack properly and I check multiple grids for the find_shortest_path method. I cannot think of how many more ways to test the code that don't feel like they are getting way more overarching then they need to be from where they are now. They cover everything that needs to be covered without getting overly specific and nitty gritty. I might not be catching an edge case I didn't consider, but overall if someone can pass my tests I can be sure that all main portions of the project are working.