Question 1

The string methods for both Deque classes run in linear time because they both initially set up an empty list that has the same length as the deque. Then, the correct values from the deque are added to the empty list, the list is joined together with commas, and then the final string has the brackets concatenated with it. The len methods for both Deque classes run in constant time because both of them just retrieve the value of self.__size. In Linked_List_Deque, push_back runs in constant time because it calls append_element from Linked_List. Push_front also runs in constant time because it calls insert_element_at exclusively for index zero, which restricts the normally linear performance of insert_element_at. Push_front will also occasionally call append_element if it is pushing into a previously empty deque, which still runs in constant time. Both pop_front and pop_back run in constant time because they call Linked_Lists's remove_element_at method, but only ever at index zero or at the tail position, so the normally linear run-time is restricted to constant time. Similarly, peek_front and peek_back run in constant time by only calling get_element_at for index zero or for the tail position, so the linear run-time of get_element_at is restricted to constant time. For Array_Deque, push_front and push_back both run in worst-case linear time because of their potential use of the __grow method. The __grow method runs in linear time because it creates a new list that has double the size of the old capacity, and then it copies over the values from the old list into the new, longer list. Pop_front and pop_back both run in constant time because only arithmetic is used to return the value and move either the front or back pointer. Likewise, peek_front and peek_back run in constant time because the front and back pointers are used to retrieve the correct value. Finally, the constructor of Array_Deque runs in constant time because it only makes basic initializations/assignments.

In Array_Deque I distinguish between an empty deque and a deque with one entry by setting the self.__front and self.__back values to None when the deque is empty. When a value is pushed onto an empty deque then I set both self.__front and self.__back to zero.

The grow method doesn't increase the array by one cell because the method performs in linear time and it would be less efficient to call it too often. So, the method doubles the size of the array so that the method will need to be called less and less often as the array gets longer.

To test these structures I first tested string, len, push_front, and push_back methods on an empty deque to check that they performed correctly. Then, I tested peek_front, peek_back, pop_front, and pop_back on an empty deque to ensure that the deque remained empty and that no errors were raised. Then, I tested push_front and push_back both together and separately with longer deques to check that they were operating on the correct side of the deque and that the grow method for the array-based deques was functioning properly. I also checked the len method with one and more calls to both push_front and push_back to make sure that these methods were correctly incrementing self.__size. Next, I tested pop_front and pop_back on deques with lengths of one and higher. I made sure that that popped value was being returned properly, that the popped value was correctly removed

from the deque, and that the len method was correctly reflecting the decrement to self.__size. Similarly, I tested peek_front and peek_back on deques with lengths of one and higher. I ensured that the correct value was being returned and that neither method was altering the length of the deque or what was contained in the deque. These test cases are complete because they test all of the methods on varying sizes of deques, and without the need for indexes as arguments, the size of the deque is the main variable that needs to be tested.

Question 2

The worst-case performance of the string method for both Stack and Queue is linear because it invokes the string method from either Array_Deque or Linked_List_Deque. The len method runs in constant time for both Stack and Queue because it relies on the constant-time len method in either Array_Deque or Linked_List_Deque. The pop and peek methods from the Stack class run in constant time because they utilize the deque methods pop_front and peek_front which both run in constant time regardless of which deque type is used. Likewise, the dequeue and peek methods from the Queue class both run in constant time because they also call the deque methods pop_front and peek_front. The enqueue method from Queue and the push method from Stack run in constant time when the Linked_List_Deque structure is used because then those methods are based in either append_element(val) or insert_element_at(val, 0) which both run in constant time. However, if Queue and Stack are based in Array_Deque, then enqueue and push will run in worst-case linear time because of the potential calls to the __grow method in Array_Deque. Lastly, the constructors of both Queue and Stack will run in constant time because they both call the function get_deque() which will either invoke the constructor from Linked_List_Deque or the constructor from Array_Deque, and both constructors run in constant time.

I think the decision to not raise exceptions doesn't really limit functionality because the only instances where an exception would be raised would be if you were to peek or pop/dequeue on an empty list. I feel as though in the majority of cases this decision works well because you can always check the length of the Stack or Queue and verify that it is or isn't empty. However, in some cases it may be helpful for the user to be alerted that they are incorrectly attempting to peek or pop/dequeue from an empty Stack or Queue.

To test the Stack and Queue structures I first tested the string, len, push, and enqueue methods on the empty structures to check that they performed correctly. Then, I tested the peek, pop, and dequeue methods on an empty Stack/Queue to ensure that these methods correctly returned none, did not alter the structure in any way, and did not raise any errors. Then, I checked the push and enqueue methods on a Stack/Queue that contained one and more values to check that they were inserting the values correctly and operating on the proper side of the structure. I also checked the len method with these test cases to ensure that the push and enqueue methods were correctly incrementing the length of their structures. Then, I tested pop and dequeue on a Stack/Queue with a length of one and greater to check that the popped value was returned correctly, that the popped value was removed from the correct side of the Stack/Queue, and that the methods were decrementing the length of the structure. Next, I also tested both peek methods on a Stack/Queue with one and more values in it to ensure that the returned value was correct, that the structure remained unaltered, and that the length of the structure stayed the same. I believe that these test cases are complete because they test all of the methods of the Stack and Queue structures. Also, the test cases test the methods on structures of different sizes to account for all of the expected implementation possibilities.

Question 3

I found that the timings for Hanoi double for each additional ring used in the game. So, the runtime of Hanoi is increasing at a rate of 2^n with n being based on the number of rings being used in the game. This performance class isn't one we have really seen before, although it seems that it is being introduced with our study of binary trees. This exponential performance is caused by the two recursive calls inside of Hanoi_rec which causes the activation record to "bounce" up and down.

I would allow a deque, stack, or queue program to use either an array-based deque or linked list-based deque by having the program in question be invoked from the command line with an additional parameter of either 0 or 1. As in Deque_Generator, the parameter 0 would correspond to a linked list-based deque implementation and the parameter 1 would correspond to an array-based deque implementation. Using the list sys.argv, I would have an if statement that would evaluate the integer that was provided on the command line and would produce the correct, corresponding deque type.

I tested Delimiter_Check by first applying it to all of the other files involved in the project to ensure that it correctly found that the delimiters were balanced. By applying Delimiter_Check to a wide range of files I was able to ensure that the delimiters would be found and checked, no matter what context they were used in. Then, I chose one file and added in extra delimiters to the code to verify that Delimiter_Check would correctly find the unbalanced delimiters. I also made sure to check that all three types of delimiters were being used in my tests. Finally, I tested Delimeter_Check with both my linked-list implementation and my array implementation.

I first tested Hanoi by keeping n=3 in the main function and checking that the printed output matched what was shown in the project description. Then, I changed the value of n and read through the printed output to ensure that all of the movements followed the two rules: that each disk was smaller than the one under it and that at the end, the disks were all placed on the destination stack. I also made sure to test Hanoi with both my linked-list implementation and my array implementation.