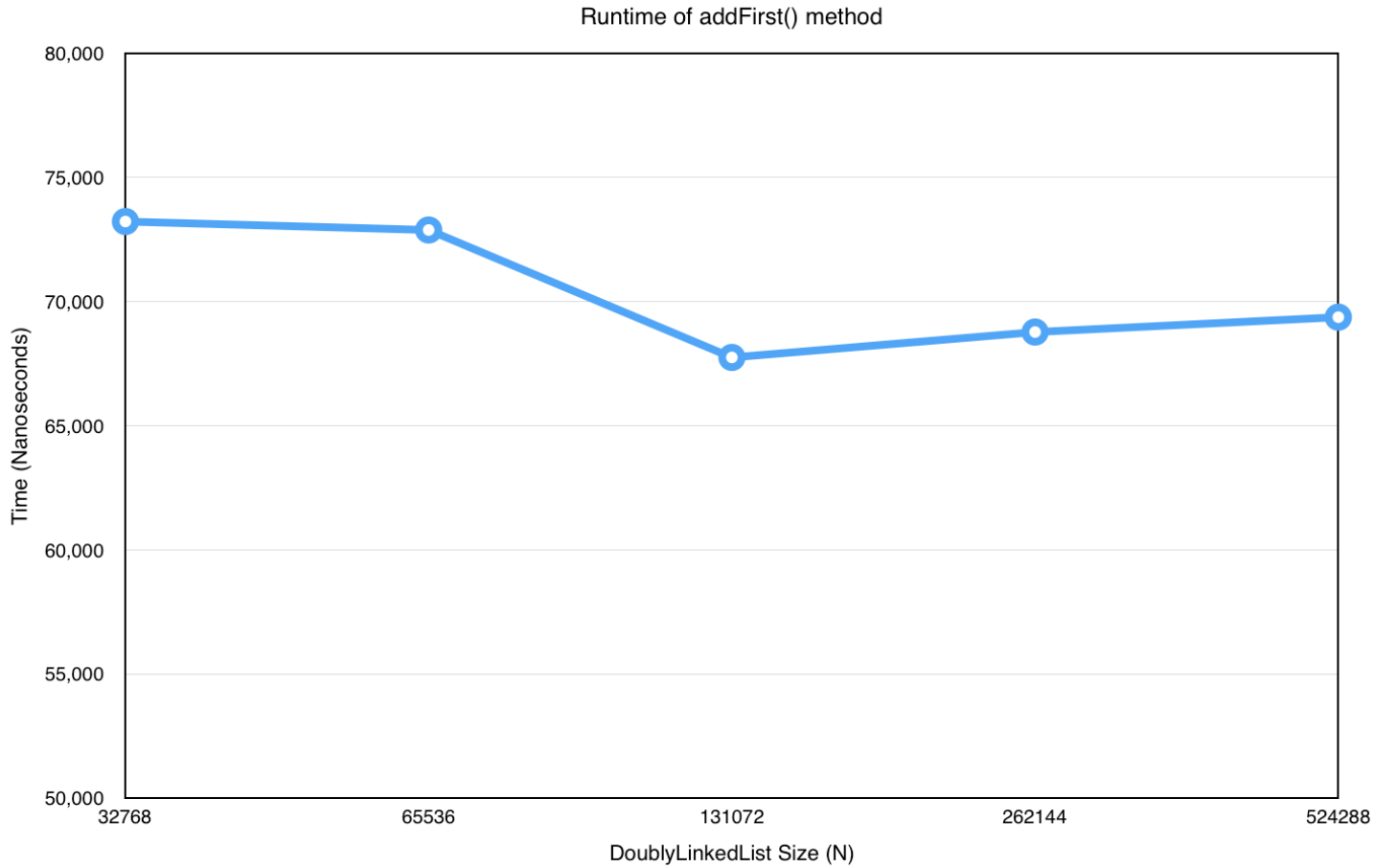
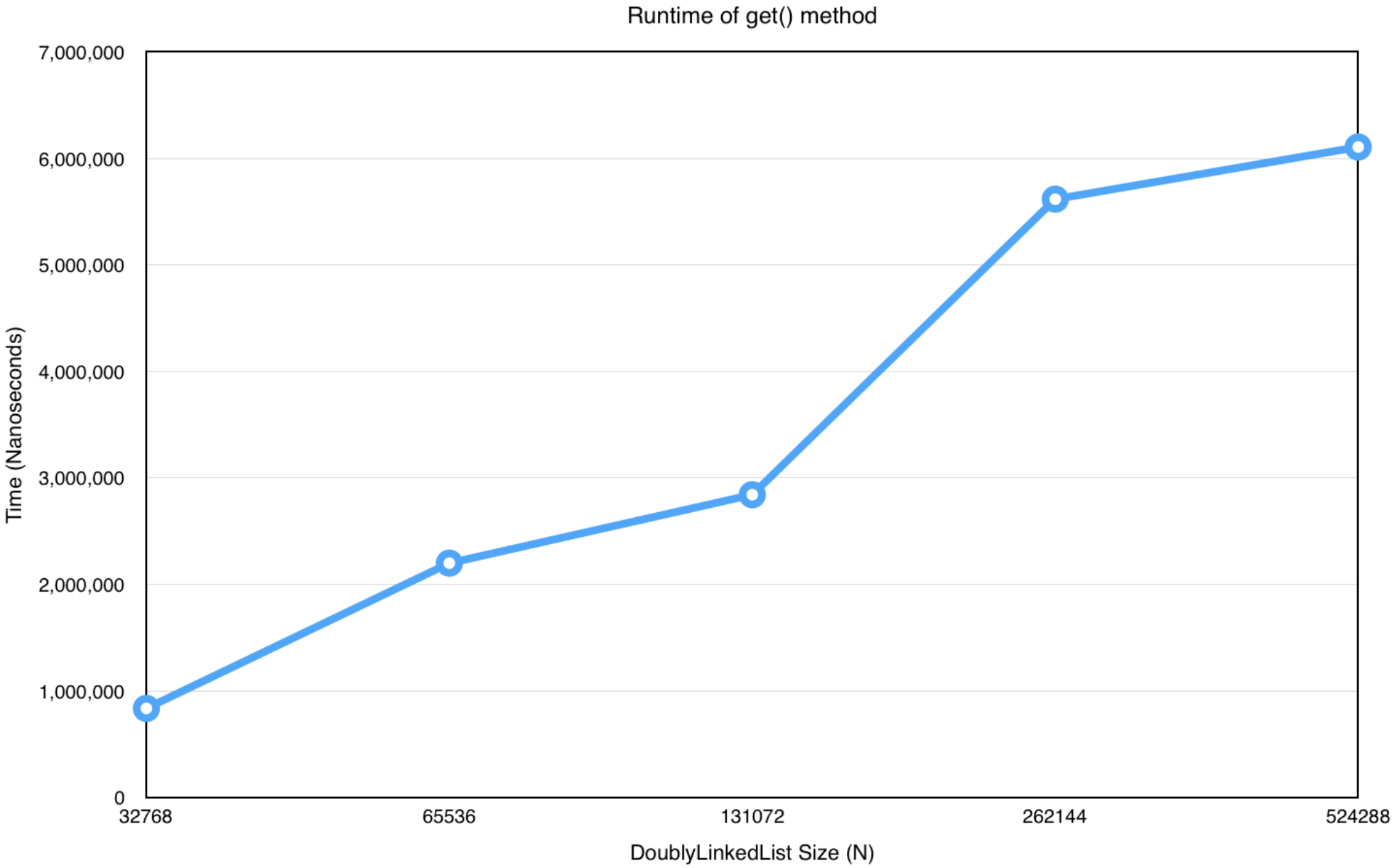
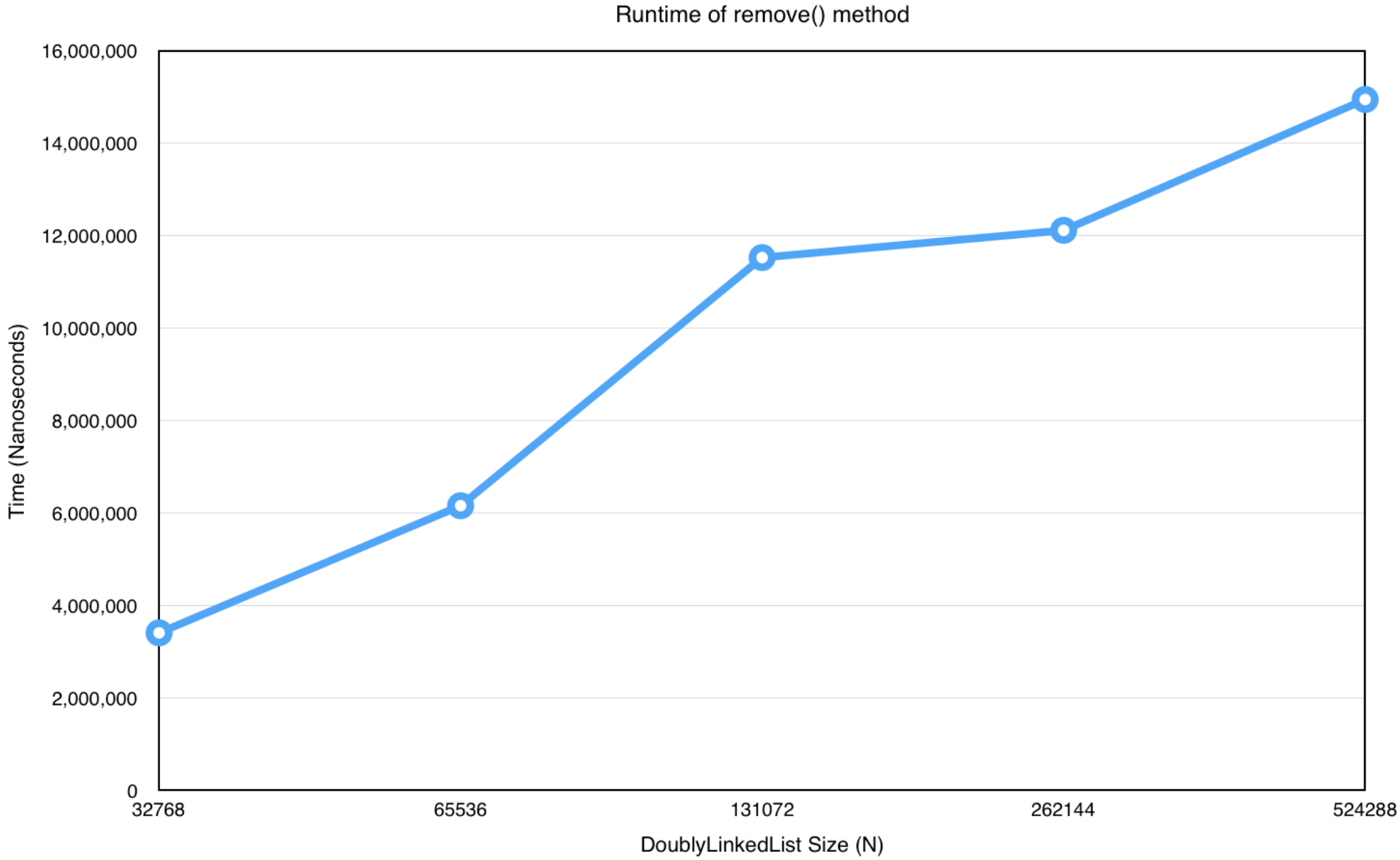
1. Collect and plot running times in order to answer each of the following questions. Note that this is this first assignment that does not specify the exact procedure for creating plots. You must design your own timing experiments that sufficiently analyze the problems. Be sure to explain all plots and answers.
Is the running time of the addFirst method O(c) as expected? How does the running time of addFirst(item) for DoublyLinkedList compare to add(0, item) for ArrayList?
Is the running time of the get method O(N) as expected? How does the running time of get(i) for DoublyLinkedList compare to get(i) for ArrayList?
Is the running time of the remove method O(N) as expected? How does the running time of remove(i) for DoublyLinkedList compare to remove(i) for ArrayList?
The run time of addFirst(item) is O(C) as expected. Since we are only adding an element onto the head of the Doubly Linked List, the N (which refers to the Doubly Linked List size) does not matter. On the other hand, add(0, item) of Java's ArrayList will be running at O(N), since we have to shift every element in the Array List to the right to make space for this new item.



The run time of get(i) is O(N) as expected. To get to our desired index, we need to start from the head (or tail) and traverse (at most) half of the Linked List. This means the worst case takes N/2 time, or O(N). Java's ArrayList runs get(i) at O(C). This is because Array List does not need to go from the head (or tail) to reach the desired index. As long as we know exactly which index, accessing the item at that index will run at O(C).



Lastly, the run time of remove(i) also displayed O(N) behavior. Again, this is because we had to start from the head (or tail) and traverse the Doubly Linked List to get to the desired index, and then remove that element. Java's ArrayList runs at O(C) at the best case, and O(N) at the worst case. The best case would be trying to remove the last element in the arraylist, since you don't have to worry about shifting any elements. The worst case would be trying to remove the very first element of the arraylist, since you need to shift every element in the arraylist to the left in order to "fill the hole".



2. In general, how does DoublyLinkedList compare to ArrayList, both in functionality and performance? Please refer to Java's ArrayList documentation.
DoublyLinkedLists do not have to worry about shifting when adding or removing elements. Simply adjusting the next node and previous node to accommodate an added element (or a removed element) is enough. This means adding or removing elements near the beginning or the end of the DoublyLinkedList will run at O(C). Java's ArrayList, however, have to shift elements forward or backwards if you are calling calling add or remove on the beginning of the ArrayList. This means adding or removing elements near the beginning of the ArrayList will run at O(N) time.

However, accessing elements given an index is faster on average with an ArrayList. This is due to DoublyLinkedLists having to start from the head node or the tail node and traverse the the LinkedList to find the desired index. The worst case would be trying to find an element at the middle of the LinkedList. On the other hand, Java's ArrayList can access a given index at O(C) time regardless of where the index is. It takes the same amount of time to access the first, middle or last element in an ArrayList.

3. In general, how does DoublyLinkedList compare to a Singly Linked List, both in functionality and performance?
While DoublyLinkedList is obviously harder to implement than a SinglyLinkedList, DoublyLinkedLists beat the SinglyLinkedLists in a number of functions. There are no difference in runtime in accessing elements in the **first half** of the LinkedList between DoublyLinkedList and SinglyLinkedList, since we will be traversing the LinkedList from the head regardless. However, if we want to access elements in the **second half** of LinkedList, DoublyLinkedLists beat SinglyLinkedLists in runtime. This is due to the tail node which is only present in a DoublyLinkedList. Using a tail node, we can traverse the LinkedList from the end instead of from the beginning.

So if we were to perform remove of the last element of the List, a DoublyLinkedList can run the method in O(C) time, whereas a SinglyLinkedList will run the method in O(N) time.

4. Compre and contrast using a LinkedList vs an ArrayList as the backing data structure for the BinarySearchSet (Assignment 3). Would the Big-Oh complexity change on add / remove / contains?
The Big-Oh complexity will have slight differences in backing the BinarySearchSet (BSS). If we called add on a BSS backed by a DoublyLinkedList (DLL), adding elements at either ends of the BSS will run at O(logN) time. Adding elements at the middle will run at O(NlogN) time. This means it is the fastest if we wanted to add either a very small element, or a very large element, these will be appended near the head or tail respectively. A BSS backed by an ArrayList (AL) however, will benefit the most only when we are adding a large element. Since this is most likely be appended to the end of the BSS, and will run at O(logN) time.

The same logic applies for remove and contains. It is better to use a DLL-backed BSS because it can benefit the most when trying to access the largest or the smallest elements, assuming the BSS is properly sorted from smallest to largest.

5. How many hours did you spend on this assignment?
I spent 7 hours on this assignment.