Problem 1:

Binary Search Analysis [10pts]

Here are two implementations of the binary search algorithm: one using recursion and the other using an iterative approach.

```
public static int recursiveBinarySearch(int[] array, int target, int left, int right) {
    if (left <= right) {
        int mid = left + (right - left) / 2;
    if (array[mid] == target)
        return mid;
    if (array[mid] < target)
        return recursiveBinarySearch(array, target, mid + 1, right);
        return recursiveBinarySearch(array, target, left, mid - 1);
    }
    return -1;
}

public static int iterativeBinarySearch(int[] array, int target) {
int left = 0;
int right = array.length - 1;
while (left <= right) {
    int mid = left + (right - left) / 2;
    if (array[mid] == target)
        return mid;
    if (array[mid] < target)
        left = mid + 1;
    else
        right = mid - 1;
    }
    return -1;
}
```

Compare the two algorithms in terms of space and time complexity. Explain your answer.

Both of the methods have the same time complexity at O(log(n)) where n is the number of elements that are in the array. This is because in a binary search algorithm, the algorithm continuously searches by dividing the search space in half with each iteration.
With space complexity, there is a difference between the recursive and iterative algorithms. Recursive binary search algorithms have to make a new array. The complexity depends on the maximum number of values in the recursion stack. When you leave the current stack, you need to call a new stack, which means that the depth is proportional to the amount that needs to be

searched. The most space complexity that a recursive binary search could return is O(log n). In iterative binary search algorithms, there is no recursion, so there is no additional space complexity due to the function being called over and over again. The are only variables that are being used, so the space complexity of an iterative search is O (1), which is constant space complexity.

Problem 2:

1. Is it possible to create an iterator on a custom object that does not implement the Iterable interface? Briefly explain why (3 points).
    - No, it is not possible to create an iterator on a custom object that does not implement the Iterable interface. This is because the Iterable interface provides a way for objects to define an iterator (something that allows for sequential access to the elements in the object). The iterator relies on methods that are defined in the interface, and there is no way for the code to iterate over the elements of the object without the interface.
2. Compare and contrast the Iterator and ListIterator interfaces in Java (7 points).
    - Iterator is an object that can be used to loop through collections. It is a universal interface that is used to traverse any collection. ListIterator is used to iterate through elements one by one on a List that implements Object. It extends the Iterator interface and is only helpful for List implemented classes. Iterator only allows for forward iteration with the method next(). ListIterator allows for both forward and backward iteration with the methods next() and previous(). ListIterator also allows for adding to a list (add()) and setting a value of the list (set()), which are features that Iterator does not have.

Problem 3:

1. Given an array of 6 elements: 2, 24, 69, 8, 1, 15. Please explain step-by-step how selection sort and insertion sort will be performed on the array. You can choose either the recursive or the iterative way (7 points).
    - Selection Sort:
        - Iteration 1: Find the minimum element in the entire array (In this case it is 1) and swap is with the first element. The array is now [1, 24, 69, 8, 2, 15]
        - Iteration 2: Find the minimum element in the array excluding the first value of the array (which is 2) and swap it with the second element. The array is now [1, 2, 69, 8, 24, 15]
        - Iteration 3: Find the minimum element in the array excluding the first and second value of the array (which is 8) and swap it with the third element. The array is now [1, 2, 8, 69, 24, 15]

        - Iteration 4: Find the minimum element in the array excluding the first, second, and third value of the array (which is 15) and swap it with the fourth element. The array is now [1, 2, 8, 15, 24, 69]
        - Iteration 5: Find the minimum element in the array excluding the values of the array that have already been sorted (which is 24). Swap this with the fifth element. The array is now [1, 2, 8, 15, 24, 69].
        - This is the last iteration as by now, all of the elements should be in their correct location. Only one element means that the element will switch spots with itself.
    - Insertion Sort:
        - Iteration 1: Starting from the second element, take it and compare with the other elements until it is in the right position from the elements that have been looked at (aka, if the 2nd number is larger than the first number keep it there, if not, move it to be the first number). The array is now [2, 24, 69, 8, 1, 15]
        - Iteration 2: Take the third element and compare with the other elements until it is in the right position from the elements that have been looked at (the first 3 elements). The array is now [2, 24, 69, 8, 1, 15].
        - Iteration 3: Take the fourth element and compare with the other elements until it is in the right position from the elements that have been looked at (the first 4 elements). The array is now [2, 8, 24, 69, 1, 15].
        - Iteration 4: Take the fifth element and compare with the other elements until it is in the right position from the elements that have been looked at (the first 5 elements). The array is now [1, 2, 8, 24, 69, 15].
        - Iteration 5: Take the sixth element and compare with the other elements until it is in the right position from the elements that have been looked at (the first 6 elements). The array is now [1, 2, 8, 15, 24, 69].

2. Why is the best case for Insertion Sort O(N)? Give an example of the scenario in which the best case occurs (3 points)
    - The best case scenario for insertion sort is O(N) because it occurs when the array is already sorted, so there is no need to change the list. This means that the time taken to sort the list is proportional to the number of elements of the list, and the list is already in the correct order so there are no changes. The iteration is trivial since the list is already in order. An example of this scenario would be [1,2,3,4,5], since there are no changes that need to be made.

Problem 4:

1. Write pseudo-code/code in Java for the following problem: Reverse a section of a singly linked list: You have a singly linked list and two integers, left and right, with left ≤ right. Your task is to reverse the nodes of the list from position left to position right and then return the modified list. For example:

   List: 12 → 15 → 24 → 10 → 19
   Left: 3
   Right: 5
   The resulting list should be: 12 → 15 → 19 → 10 → 24

Java
```java
public LinkedList ReverseLinkedListInSections(LinkedList modiList,int left, int
right) {
	Node head = modiList.getFirstNode();
	if (head == null || left == right) {
		return modiList;
	}
	//in case of left being 1, will be left out later in the loop
	Node placeholder = new Node(-25, null);
	placeholder.setNext(head);


	Node beforeLeft = placeholder;
	//Loop for getting the Node right before the left position
	for (int i = 1; i < left; i++) {
		beforeLeft = beforeLeft.getNext();
	}

	Node beforeRight = beforeLeft;
	//Loop for getting the last Node that needs to be flipped
	for (int i = left; i <= right; i++) {
		beforeRight = beforeRight.getNext();
	}

	Node leftNodeFlip = beforeLeft.getNext();
	Node rightNodeFlip = beforeRight.getNext();

	beforeRight.setNext(null);

	Node oldNode = null;
	Node newNode = leftNodeFlip;
	Node nextNode = newNode.getNext();

	while (newNode != null) {
		nextNode = newNode.getNext();
```

```
            newNode.setNext(oldNode);
            oldNode = newNode;
            newNode = nextNode;
        }

        beforeLeft.setNext(oldNode);
        leftNodeFlip.setNext(rightNodeFlip);

        return modiList;
    }
```