Daniel Ye, David Siher

1.
    a.) The specification for add(T x) does not say that x is added to the set in a sorted order.
    /**Adds x to the set if not already there while keeping the list in ascending order. */
    b.) The specification for contains(T x) is fine.
    c.) Remove should specify what happens if if x is null or not in the SortedSet.
    /**Removes element x if it is in the set and is not null.*/
    d.) First() does not specify what happens if the set is empty.
    /**Returns the first element of the set unless the set is empty, in which case it returns null. */
2.

```
/**Returns the first element of the set unless the set is empty, in which case it returns
null. */
T first() {
        return head.value;
}
/**Removes element x if it is in the set and is not null.*/
void remove (T x) {
        ListNode<T> prev = head;
        ListNode<T> current = prev.next;
        If (contains x) {
                return;
        }
        while (!current.value.equals(x)) {
                prev = prev.next;
                current = current.next;
        }
        prev.next = current.next;
}
```

3.

```
/**Returns the first element of the set unless the set is empty, in which case it returns
null. */
T first() {
        return head.value;
}
/**Removes element x if it is in the set and is not null.*/
void remove (T x) {
        ListNode<T> prev = head;
        ListNode<T> current = prev.next;
        If (contains x) {
                return;
        }
```

```
		while (!current.value.equals(x)) {
			prev = prev.next;
			current = current.next;
		}
		prev.next = current.next;
	}
	/**Adds x to the set if not already there. */
	void add (T x) {
		ListNode<T> temp = new ListNode<T>(T, null);
		if (contains(x)) {
			return;
		}
		If (head == null) {
			head = temp;
		} else {
			head.next = head;
			head = temp;
		}
	}
```

4. The advantage of UnsortedList is that when adding a new element, the program does not have to iterate through the list to find the appropriate position for the element being added in order to maintain ascending order, meaning the complexity of add(T x) is always O(1). For SortedList, remove(T x) can be faster because the program has some information about where x is stored in the list based on its value, allowing for faster searching (such as with a binary search) for the element that needs to be removed. An UnsortedList therefore is better when used for a function where the user mostly adds elements to the list without removing or searching for as many, while a SortedList is more efficient when a user is more focused on being able to locate elements in the list and manipulate them, since the ordered nature of the list allows for quicker searching of certain elements.

Loop Invariants

```
/**
 * Returns the largest index i such that a[i] == k.
 * Requires: k is in a, and a is sorted in ascending order
 */
int search(int[] a, int k) {
	int l = 1, r = a.length          ;
	 while (l < r) {
		int m = (l+r)/2;
		if (k < a[m]) r = m;
		else l = m + 1;
	}
	return l - 1;
}
```

B. The post condition of the loop is that l - 1 is the largest index such that a[l - 1] == k.
C.

1. $1 \leq l \leq r \leq a.length$
2. $k \in a[l-1...r-1]$

D. **Initialization:**

1. a.length is at least 1 by requiring that k be in the array, and l is initialized to 1 while r is initialized to a.length, therefore $1 \leq l \leq r \leq a.length$ is true.
2. When first initialized $a[l - 1...r - 1]$ spans the whole array, which means k must be there by the precondition.

Preservation:

m is the average of l and r rounded down, so $l \leq m < r$ because m cannot equal r without breaking the loop guard ($l < r$). This means that either $k \in a[l-1...m]$ or $k \in a[m+1...r-1]$

- **Case** $k \in a[l-1...m]$
  $k \leq a[m]$ must be true because the array is sorted. The loop body then sets $r' = m$ and $l' = l$. Because $1 \leq l \leq m < r \leq a.length$ and $k \in a[l-1...m-1]$, it becomes $1 \leq l' \leq r' \leq a.length$ and $k \in a[l'-1...r'-1]$ matching the loop invariant.
- **Case** $k \notin a[l-1...m]$
  Because $k \in a[l-1...r-1]$, then $k \in a[m+1...r-1]$. Also, $k \geq a[m]$ because the array is required to be sorted. The loop body then sets $r' = r$ and $l' = m + 1$. Since $1 \leq l < m + 1 \leq r \leq a.length$ and $k \in a[m+1..r-1]$, it means $1 \leq l' \leq r' \leq a.length$ and $k \in a[l'-1...r'-1]$ again matching the loop invariant.

Postcondition:

For the algorithm to be correct, $a[l-1] = k$. If the loop guard is broken, $l \geq r$, but the invariant guarantees that $l - 1 \leq r - 1$ so $l = r$. The invariant also says $k \in a[l-1...r-1]$ which is now a single element, so k must be at $a[l-1]$.

Termination:

By the first part of the invariant, $r - l$ is guaranteed to be nonnegative. In the case where $k \in a[l-1...m]$, $m < r$, so $r' - l' < r - l$. In the other case, we know $l < m + 1$, so, $r' - l' < r - l$. This means that $r - l$ gets smaller on every loop iteration as long as $l < r$, and eventually terminates with $l = r$.

5. No, because in order for sin(n) to be O(cos(n)), there must be some k constant where sin(n) is always less than k*cos(n). However, no matter what the value of k is, all values of n in the set n = π/2 + πr where r is an integer give sin(n) to be 1 and k*cos(n) to be zero for all values of k. This means that an n_0 doesn't exist where sin(n) will always be greater than k*cos(n) when n is greater than some n_0 since eventually n will cycle around to being a part of π/2 + πr. Therefore, sin(n) is not always less than k*cos(n) for any constant k so sin(n) cannot be described as O(cos(n)).
6. No, because $2^{2^\wedge n}$ is always larger than k * 4^n at a certain value of n for any constant k. We see this by taking $\log_2$ of both sides, yielding $2^n < \log_2 k*4^n$. By distributing, we get $2^n < \log_2 k + \log_2 4^n$ which is equivalent to $2^n < \log_2 k + 2n$. No matter how large k is, 2^n will

eventually become larger than $\log_2 k + 2n$ as n increases, so there is no n_0 where all $n > n_0$ hold that $2^{2^n} < k*4^n$.