Practical 8

A. Algorithms for Searching

1. When searching a sorted array sequentially, you can ascertain that a given item does not appear in the array without searching the entire array. Modify the method **contained** in the Chapter7\array\SortedArrayList class such that it takes advantage of this observation when searching a sorted array sequentially.

```
2, 4, 5, 6, 9, 10

boolean found = arrayList.contains(7);
```

1) Found it
2) Reach the end of the array
3) greater

compareTo equal 0, greater 1, less than -1

array[i].compareTo(?)
anEntry.compareTo(?)

//Dih Yong

```
public boolean contains(T anEntry) {
      boolean found = false;
      for (int index = 0; !found && (index < length); index++) {
            if (anEntry.equals(array[index])) {
                  found = true;
            }
            else if(anEntry.compareTo(array[index])<1){
                  break;
          }
      }
      return found;
}
//modify
```

2. Write a new array implementation of the sorted list as follows:

- The class implements the SortedListInterface from Chapter 7.
- Provide an iterative version of the **binary** search algorithm that will return the index of the target entry in the array. If the array does not contain the target entry, the method should return the index of the insertion point as a negative value.
- Implement the **contains**, **add** and **remove** methods such that they invoke the binary search *method* to determine the index of the given entry in the array.

length=1
l=0
r=0
m=0
m = (3 + 3)/2 = 3
desiredItem = 3

```
 //Yong Chen - binary search
private int binarySearch(T arr[], T desiredItem)
    {
        int left = 0, right = arr.length-1;
        while (left  <= right ) {
            int middle = (left  + right ) / 2;

            if (desiredItem.equals(arr[middle]))
                return middle ;
            if (desiredItem.compareTo(arr[middle]) < 0)
                right = middle  - 1;
            else
                left = middle  + 1;

        }
        return -(left + 1);
    }
```

```
//Joan - contains
public boolean contains(T entry) {
    return binarySearch(entry) >= 0;
  }
```

```
2, 4 , 5 , 6, 8
newEntry = 3
```

```
//Yong Kit - add
public boolean add(T newEntry) {
    int search = binarySearch(newEntry);
    makeRoom(search + 1);
    array[search] = newEntry;
    length++;
    return true;
}

private void makeRoom(int newPosition) {
    int newIndex = newPosition - 1;
    int lastIndex = length - 1;
```

```
        for (int index = lastIndex; index >= newIndex; index--) {
          list[index + 1] = list[index];
        }
      }


    //remove (Yong Kit)
    public boolean remove(T anEntry) {

        int search = binarySearch(anEntry);
        if(search >= 0){
            removeGap(search);
            length--;
            return true;
        }
        else
            return false;
    }
```

B. Algorithms for Sorting

3. Implement the iterative **selection** sort algorithm as a static method in the SortArray class.

   3, 2 , 5, 6 , 1 , 4

1

//Hao Han

```
public static void sort(int[] a)
// Sort the contents of array a in ascending numerical order
{
 for(int i=0; i<a.length-1; i++)
 {
     int temp,pos=i;
     for(int j=i+1; j<a.length; j++){
        if(a[j] < a[pos])
            pos = j;
     }
     //swap the elements indexed pos1 and pos2 within array a
     temp = a[i]; //swap
     a[i] = a[pos];
     a[pos] = temp;

  }
}
```

4. Implement the iterative **insertion** sort algorithm (including insertInOrder) as Java methods.

   //Jun Yan

```
public static <T extends Comparable<? super T>> void insertionSort(T[] a){
        T valueToInsert = null;
        int n = a.length;

        for(int i = 1; i < n; i++){
            valueToInsert = a[i];
            insertInOrder(a, valueToInsert, i-1);
        }
}

private static <T extends Comparable<? super T>> void insertInOrder(T[]
arr, T element, int indexOfLastInSortedRegion){

            //find the right position to insert the value
             //into the sorted region
            int index = indexOfLastInSortedRegion;
            while((index>=0) && (arr[index].compareTo(element) > 0)){
                //move the value in the array to the back

                    arr[index+1] = arr[index];
                    index--;
            }

            //insert the value into the position found
            arr[index+1] = element;
    }
}
```

5. Implement the **recursive** selection sort algorithm as a private method using the following method header:

```
//JiaJian
//Yong Kang
void selectionSort(T[] a, int first, int last){
    if(last>first)
    {
         int temp,pos=first;
         for(int j=first+1; j<a.length; j++){
            if(a[j] < a[pos])
                pos = j;
         }
         //swap the elements indexed pos1 and pos2 within array a
         temp = a[first]; //swap
         a[first] = a[pos];
         a[pos] = temp;
          selectionSort(a,first+1,last);
      }
}
```

and include the following public method to invoke the recursive method:

```
void selectionSort (T[] a, int n){ //n = length
      selectionSort(a , 0, n-1);
}
```