COMP132 – Computer Science II
Fall 2015

Homework #10
Sorting and Searching 2
Solutions

1. The `sort` method in the `comp132.examples.sorting.MergeSort` class created new sub-arrays for the left and right parts of the array. Creating these arrays for each recursive call takes time and creates an unnecessary inefficiency in the program. If we use a recursive problem transformation to introduce a new helper method this inefficiency can be removed. The class below shows how the `sort` method might be transformed by adding parameters for `start` and `end`, which indicate the portion of the array to be sorted. It also gives the signature of a transformed `merge` method. Assuming the new `merge` method works, give an implementation of the new `sort` method.

```
public class MergeSortWithHelper {
   /**
    * Merge sort the provided array.
    *
    * @param arr the array to be sorted.
    * @return the sorted array.
    */
   public static int[] sort(int[] arr) {
      return sort(arr, 0, arr.length);
   }

   /*
    * Sort the contents of arr from index start, up to
    * but not including end.
    */
   private static int[] sort(int[] arr, int start, int end) {
      // Give an implementation of this method.
   }

   /*
    * Merge the contents of the array from start up to but not
    * including mid with the contents from mid up to but not
    * including end.  Assume that [start…mid) and [mid…end) are
    * in sorted order. Return arr with the indicated sections merged.
    */
   private static int[] merge(int[] arr, int start, int mid, int end){
      // Assume this method works.
   }
}
```

```java
/*
 * Sort the contents of arr from index start, up to but not
 * including end.
 */
private static int[] sort(int[] arr, int start, int end) {

  if (end - start == 1) {
    return arr;
  }
  else {
    // Sort the left half of the array.
    sort(arr, start, (start + end) / 2);

    // Sort the right half of the array.
    sort(arr, (start + end) / 2, end);

    // Merge the two sorted halves into a sorted array.
    return merge(arr, start, (start + end) / 2, end);
  }
}
```

2. The linear search algorithm is defined as:

search(arr, x): return the index at which x is found in arr
or -1 if x is not contained in arr.

If we add a parameter indicating the index at which to start the search it becomes possible to give a recursive definition for linear search.

search(arr, x, start): return the index at which x is found in arr
or -1 if x is not contained in the portion of arr from index start to the end of the array.

Give a recursive definition of linear search using this second definition of the search method. Clearly state the base case(s) and the recursive case(s). Do not implement your definition in Java.

Base Cases:	start == arr.length	return -1
	arr[start] == x	return start
Recursive Case:	return search(arr, x, start+1)	

3. In class we developed the following recursive definition for binary search:

```
search(arr, x, start, end)
  Let mid = (start+end) / 2

  Base Cases: start>end        return -1
              arr[mid] == x    return mid

  Recursive Cases: arr[mid] > x   return search(arr, x, start, mid-1)
                   arr[mid] < x   return search(arr, x, mid+1, end)
```

Consider the following sorted list of integers:

```
4   6   9  10  12  15  21  28  33  45  49  50 58
```

The first value in this list that would be inspected by our binary search algorithm is 21. Give the full sequence of values that would be inspected by the our binary search algorithm when searching for each of the following values:

a. 49

21, 45, 50, 49

First inspection:	start = 0, end = 12, mid = (0+12) / 2 = 6	arr[6] = **21**.
Second inspection:	start = 7, end = 12, mid = (7+12) / 2 = 9	arr[9] = **45.**
Third inspection:	start = 10, end = 12, mid = (10+12) / 2 = 11	arr[11] = **50.**
Fourth inspection:	start = 10, end = 10, mid = (10+10) / 2 = 10	arr[10] = **49.**

b. 7

21, 9, 4, 6

First inspection:	start = 0, end = 12, mid = (0+12) / 2 = 6	arr[6] = **21**.
Second inspection:	start = 0, end = 5, mid = (0+5) / 2 = 2	arr[2] = **9.**
Third inspection:	start = 0, end = 1, mid = (0+1) / 2 = 0	arr[0] = **4.**
Fourth inspection:	start = 1, end = 1, mid = (1+1) / 2 = 1	arr[1] = **6.**
Fifth inspection:	start = 2, end = 1	

start > end now, so 7 is not in the array and the method returns -1.

4. Consider the following unsorted list of integers:

```
45   33   4   6   28   10   50   9   12   58   15   21   28
```

a. Give the full sequence of values that would be inspected by our binary search algorithm if it were used to search this list for the value 9.

50, 4, 28, 10

First inspection:	start = 0, end = 12, mid = (0+12) / 2 = 6	arr[6] = **50**.
Second inspection:	start = 0, end = 5, mid = (0+5) / 2 = 2	arr[2] = **4.**
Third inspection:	start = 3, end = 5, mid = (3+5) / 2 = 4	arr[4] = **28.**
Fourth inspection:	start = 5, end = 5, mid = (5+5) / 2 = 5	arr[5] = **10**
Fifth inspection:	start = 6, end = 5	

start > end now, so binary search incorrectly concludes that 9 is not in the array and the method returns -1.

b. Explain why binary search requires that the array be in sorted order.

At each step binary search eliminates half of the remaining list from further consideration. In doing so binary search makes the implicit assumption that the value for which it is searching cannot possibly appear in the eliminated half of the list. This assumption is only valid if the list is in sorted order. For example, in part a above, the first comparison made by binary search eliminated the right half of the list. But because the list was unsorted there was no guarantee that the value 9 would be in the left half of the list. Thus, the implicit assumption that the value was not contained in the eliminated portion of the list was not satisfied, and the value was not found. If on the other hand the array had been sorted then the value 9 would have been guaranteed to be in the left half, the implicit assumption would have been satisfied, and the value would have been found.

5. Modify the `binarySearch` and `binarySearchHelper` methods in the `comp132.examples.sorting.Search` class so that they can search an array of `Student` objects (the `Student` class is defined in the `comp132.examples.sorting.java` package). Copy and paste the modified methods as your answer to this question.

The only change that must be made is to use the `.compareTo` method defined in the `Student` class instead of == and > or <.

```
public static int binarySearch(Student[] arr, Student value) {
   return binarySearchHelper(arr, value, 0, arr.length-1);
}

private static int binarySearchHelper(Student[] arr, Student value,
                                      int low, int high) {
   if (low > high) {
      return -1;  // value not found.
   }
   else {
      int mid = (low + high) / 2;

      if (arr[mid].compareTo(value) == 0) {
         // arr[mid] is the same as value – Found it!
         // NOTE: could also use .equals here
         return mid;
      }
      else if (arr[mid].compareTo(value) < 0) {
         // arr[mid] comes before value, so search right half.
         return binarySearchHelper(arr, value, mid+1, high);
      }
      else {
         // arr[mid] comes after value, so search left half.
         return binarySearchHelper(arr, value, low, mid-1);
      }
   }
}
```

NOTE: It is worth noting that if the type `Student` in the method signature were replaced with the interface `Comparable`, then this method would be able to search an ordered list of any class that implements the `Comparable` interface. That is exactly what is done in the `Arrays.binarySearch` and the `Collections.binarySearch` methods provided with Java!

6. Modify the `compareTo` method for the `Student` class (from the `comp132.examples.sorting.java` package) so that `Students` will be sorted in order by graduation year and alphabetically within graduation year. So the first student in the list will be at the beginning of the alphabet and have the smallest graduation year. Copy and paste your modified compareTo method as your solution to this question.

```
public int compareTo(Student stu) {
   if (this.equals(stu)) {
      return 0;   // Same student!
   }
   else if (gradYear != stu.getGradYear()) {
      return gradYear - stu.getGradYear();
      // NOTE: is < 0 if gradYear is comes before stu.gradYear.
   }
   else {
      // Grad years are the same so order alphabetically.
      // Use the compareTo method from String to do this.
      return this.name.compareTo(stu.getName());
   }
}
```

7. Sometimes a class implements the `compareTo` method, but it is necessary to sort the items into an order that is different than the one dictated by that method. In Java the `Comparator` interface and the two-argument `Collections.sort` method provide the mechanism for doing this. You can read about the `Comparator` interface and the two-argument `Collections.sort` method in the JavaDoc. Define a class that implements `Comparator` that will sort a list of `Student` objects into order by increasing graduation year and by increasing GPA within graduation years. Two students graduating in the same year with the same GPA should be ordered alphabetically.

```
public class YearGPAOrdering implements Comparator<Student> {

    public int compare(Student o1, Student o2) {
        if (o1.getGradYear() < o2.getGradYear()) {
            return -1;      // o1 graduates before o2
        }
        else if (o1.getGradYear() > o2.getGradYear()) {
            return 1;       // o1 graduates after o2
        }
        else {
            // o1 and o2 graduate in the same year.
            if (o1.getGpa() < o2.getGpa()) {
                return -1;  // o1 has a lower gpa than o2
            }
            else if (o1.getGpa() > o2.getGpa()) {
                return 1;   // o1 has a higher gpa than o2
            }
            else {
                // o1 and o2 have the same gpa.
               return o1.getName().compareTo(o2.getName());
            }
        }
    }
}
```

Bonus: Give an implementation of the transformed `merge` method as defined in problem #1.

```
private static int[] merge(int[] arr, int start, int mid, int end) {
   int leftIndex = start;  // index of next value to use in left part
   int rightIndex = mid;   // index of next value to use in right part

   // Temporary working array to hold merged values.
   int[] merged = new int[end - start];

   // The index at which to place next value in the merged list.
   int mergedIndex = 0;

   // Keep going until the segments have been fully merged…
   while (mergedIndex < merged.length) {

      if (leftIndex < mid && rightIndex < end) {
         // Still have values in both the left part: arr[start…mid)
         // and the right part: arr[mid…end)

         if (arr[leftIndex] < arr[rightIndex]) {
            // Next value in the left part is smaller so use it.
            merged[mergedIndex] = arr[leftIndex];
            leftIndex++;
         }
         else {
            // Next value in the right part is smaller so use it.
            merged[mergedIndex] = arr[rightIndex];
            rightIndex++;
         }
      }
      else if (leftIndex < mid) {
         // All values in right part: arr[mid…end) have been used,
         // so take next one from the left part: arr[start…mid).
         merged[mergedIndex] = arr[leftIndex];
         leftIndex++;
      }
      else {
         // All values in left part: arr[start…mid) have been used,
         // so take next one from the right part: arr[mid…end).
         merged[mergedIndex] = arr[rightIndex];
         rightIndex++;
      }

      mergedIndex++;
   }

   // Copy the merged list back into place in the original array.
   for (int i=start; i<end; i++) {
      arr[i] = merged[i-start];
   }

   return arr;
}
```