Name: SOLUTIONS
Student ID: ________________

<table>
<thead>
<tr>
<th>Question</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(15)</td>
</tr>
<tr>
<td>2</td>
<td>(20)</td>
</tr>
<tr>
<td>3</td>
<td>(20)</td>
</tr>
<tr>
<td>4</td>
<td>(15)</td>
</tr>
<tr>
<td>5</td>
<td>(15)</td>
</tr>
<tr>
<td>6</td>
<td>(15)</td>
</tr>
</tbody>
</table>

NOTE: Any questions on writing code must be answered in Java using Data Structures topics covered in lectures 1-23
1. [15 pts]

a) Many operations can be performed faster on sorted data than on unsorted data. For each of the following operations, state whether it could be performed faster if the data values were sorted (do not take the cost of the sorting into account). Please give a brief explanation. Assume an array is used to store data. [8 pts]

   i) Find the median value.

   Yes, Binary Search takes $O(\log n)$ time to find the median value in an sorted array. Takes $O(n)$ in an unsorted array.

   ii) Calculate the sum of all data in the array.

   No, $O(n)$ time no matter sorted or unsorted data.

   iii) Partition the array according to a given pivot value.

   Yes, takes $O(\log n)$ time on a sorted array. $O(n)$ time on an unsorted array.

   iv) Insert a new value to the middle position (which is empty) of the array.

   No, both take $O(1)$ time.
b) Sort the array below to ascending order (left-to-right) using **Insertion Sort**. Note: You do not need to write any code, but you do need to show each step in the sorting process for full credit. [3 pts].

<table>
<thead>
<tr>
<th>13</th>
<th>2</th>
<th>9</th>
<th>42</th>
<th>20</th>
<th>31</th>
<th>7</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>2</td>
<td>9</td>
<td>42</td>
<td>20</td>
<td>31</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>9</td>
<td>42</td>
<td>20</td>
<td>31</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>13</td>
<td>42</td>
<td>20</td>
<td>31</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>13</td>
<td>20</td>
<td>42</td>
<td>31</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>13</td>
<td>20</td>
<td>31</td>
<td>42</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>9</td>
<td>13</td>
<td>20</td>
<td>31</td>
<td>42</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>20</td>
<td>31</td>
<td>42</td>
</tr>
</tbody>
</table>


c) Sort the array below to ascending order (left-to-right) using **Quick Sort**. Please use the last element in each array/sub-array as the pivot and highlight all the pivots. Note: You do not need to write any code, but you do need to show each step in the sorting process for full credit. [4 pts].

<table>
<thead>
<tr>
<th>45</th>
<th>16</th>
<th>8</th>
<th>32</th>
<th>10</th>
<th>6</th>
<th>33</th>
<th>29</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>8</td>
<td>10</td>
<td>6</td>
<td>29</td>
<td>45</td>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>8</td>
<td>10</td>
<td>29</td>
<td>45</td>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>8</td>
<td>10</td>
<td>29</td>
<td>32</td>
<td>33</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>10</td>
<td>16</td>
<td>29</td>
<td>32</td>
<td>33</td>
<td>45</td>
</tr>
</tbody>
</table>
2. [20 pts]

a) As described in the class, Insertion Sort sequentially traverses an array when making comparisons to find the proper place for the element that is currently being processed. Please write a method BinaryInsertionSort() that instead uses Binary Search to find the proper place for the current element. Please also write the associated BinarySearch() method. The sorted array should be in **descending order**.

**[15 pts]**

```java
public int BinarySearch(int[] searchArray, int targetPos)
{
    int index=-1, low=0, high = targetPos-1;
    int middle = 0;
    while( low<=high )
    {
        middle = (low+high)/2;   // middle position
        if( searchArray[middle]==searchArray[targetPos] )
        {
            index = middle;
            return index;
        }
        else if( searchArray[middle]>searchArray[targetPos] )
            low = middle + 1;  // continue search first half
        else
            high = middle - 1;  // continue search second half
    }
    index = low;
    return index;
}
```

```java
public void BinaryInsertionSort(int[] myArray)
{
    int temp=0, i=0, j=0;
    for( i=1; i<myArray.length; i++ )
    {
        // find the proper position
        int pos = BinarySearch(myArray, i);
        temp = myArray[i];
        // move the elements after the found position
        for( j=i; j>pos; j-- )
        {
            myArray[j] = myArray[j-1];
        }
        myArray[j] = temp;  // insert the current element
    }
}
```

**b) Is this modified Insertion sort algorithm quicker than the original Insertion Sort method for most array sizes? Why or Why not? [5 pts]**
It is quicker because the time to find the proper insert position is reduced. But the big-O complexity is still $O(n^2)$ since those elements after the found position still need to be shifted.
3. [20 pts]

You are given some methods in a **Quick Sort** algorithm that sorts an **ArrayList** into **descending order** (the biggest values are at the beginning of the ArrayList). The Quick Sort algorithm can be implemented using three methods: QuickSort, the initial call for the method, recQuickSort, which recursively breaks up and sorts the array, and partitionIt(), which determines the partitioning of each subarray. You are given the QuickSort() and the recQuickSort() method for this algorithm, but not the partitionIt() method. Please write the partitionIt() method for this algorithm. The pivot should always be the first element in each array/sub-array. Please briefly comment your code. The following methods in the ArrayList class could be used.

- **boolean add(int element)**
  Appends the specified element to the end of this list.
- **int remove(int index)**
  Removes the element at the specified position in this list.
- **int get(int index)**
  Returns the element at the specified position in this list.
- **int set(int index, int element)**
  Replaces the element at the specified position with the specified element.
- **int size()**
  Returns the number of elements in this list.

```java
public void QuickSort(ArrayList<Integer> myArray) {
    recQuickSort(0, myArray.size()-1);
}

public void recQuickSort(int left, int right) {
    if(right-left <= 0)  // if size <= 1,
        return;  // already sorted
    else  // size is 2 or larger
    {
        // leftmost item
        int pivot = (Integer) myArray.get(left);
        // partition range
        int partition = partitionIt(left, right, pivot);
        recQuickSort(left, partition-1);  // sort left side
        recQuickSort(partition+1, right);  // sort right side
    }
}

public int partitionIt(int left, int right, int pivot) {
    int leftPtr = left;  // left+1 (after ++)
```
int rightPtr = right+1;  // right (after --)
while(true)
{
    // find smaller item, make sure leftPtr+1 is valid
    while( leftPtr+1 < myArray.size() && myArray.get(++leftPtr) > pivot )
    {
        // find bigger item
        while( rightPtr > 0 && myArray.get(--rightPtr) < pivot )
        {
            if(leftPtr >= rightPtr)  // if pointers cross,
                break;                    // partition done
            else  // not crossed, so
                swap(leftPtr, rightPtr);  // swap elements
        }
    }
    swap(rightPtr, left);  // restore pivot
    return rightPtr;      // return pivot location
}

public void swap(int dex1, int dex2)  // swap two elements
{
    int temp = myArray.get(dex1);  // A into temp
    myArray.set(dex1, myArray.get(dex2));  // B into A
    myArray.set(dex2, temp);  // temp into B
}
4. [15 pts]

You are given some methods in a modified Merge Sort algorithm that sorts an input integer array into **descending order** while separating odd integers from even integers. Even integers should be placed before odd integers in the sorted array. Here is an example:

Unsorted array: 1 2 3 4 5 6 7 8  
Sorted array: 8 6 4 2 7 5 3 1

You are given the high level ModfiedMergeSort() method and the recursive recMergeSort() method which breaks up the array, as shown below. Please write the merge() function for this algorithm. Please briefly comment your code to make it easy to understand.

```java
public void ModifiedMergeSort(int[] myArray)
{
    // provides workspace
    int[] workSpace = new int[myArray.length];
    recMergeSort(workSpace, 0, myArray.length-1);
}

private void recMergeSort(int[] workSpace, int lowerBound, int upperBound)
{
    if(lowerBound == upperBound) // if range is 1,
        return;  // no use sorting
    else
    {
        int mid = (lowerBound+upperBound) / 2;  // find midpoint
        // sort low half
        recMergeSort(workSpace, lowerBound, mid);
        // sort high half
        recMergeSort(workSpace, mid+1, upperBound);
        // merge them
        merge(workSpace, lowerBound, mid+1, upperBound);
    }
}

private void merge(int[] workSpace, int lowPtr, int highPtr, int upperBound)
{
    int j = 0;  // workspace index
    int lowerBound = lowPtr;
    int mid = highPtr-1;
    int n = upperBound-lowerBound+1;  // # of items

    while(lowPtr <= mid && highPtr <= upperBound)
    {
        // if same odd or even, compare the value
    }
```
// even is greater than odd
// descending order
if (( (myArray[highPtr] > myArray[lowPtr] ) &&
( myArray[highPtr]%2 == myArray[lowPtr]%2 ) ) ||
( myArray[highPtr]%2 == 0 && myArray[lowPtr]%2 == 1 ) )
workSpace[j++] = myArray[highPtr++];
else
workSpace[j++] = myArray[lowPtr++];

// copy the rest of the elements to workSpace[]
while(highPtr <= upperBound)
workSpace[j++] = myArray[highPtr++];
// copy the rest of the elements to workSpace[]
while(lowPtr <= mid)
workSpace[j++] = myArray[lowPtr++];

// copy back to theArray
for(j=0; j<n; j++)
myArray[lowerBound+j] = workSpace[j];
}
5. [15pts]
The following questions deal with Binary Search Tree. All the nodes of the Binary Search Tree are defined using the following class:

```java
class Node {
    public int Key;
    public String Value;
    public Node Right;
    public Node Left;
}
```

a) Insert items with the following keys (in the given order) into an initially empty binary search tree: 50, 40, 30, 46, 38, 58, 56, 52. Draw the tree after you have inserted all items (just show the key of each node). [5 pts]

```
50
  
40 58
  
30 46 56
  
38 52
```

b) Write a method findSize() that returns the total number of nodes in an arbitrary binary search tree. Please indicate the complexity of your method in big-O notation. [10 pts]

```java
public int findSize(Node root) {
    int count = 1;
    if (root == null) // if node is empty
        return 0;
    // add the number of nodes in left subtree
    count += findSize(root.Left);
    // add the number of nodes in right subtree
    count += findSize(root.Right);
    return count;
}
```

The complexity of this method is O(n), in which n is the total number of nodes in this tree.
6. [15 pts]

Given the following numbers \{71, 23, 73, 97, 44, 77, 87, 31\} and a hash function \( h(x) = (x \mod 10) \):

a) Insert these numbers into an initially empty hash table using linear probing. Show the populated hash table. Assume the array that stores this hash table has 10 cells. [7 pts]

<table>
<thead>
<tr>
<th>Slot</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>71</td>
<td>31</td>
<td>23</td>
<td>73</td>
<td>44</td>
<td></td>
<td>97</td>
<td>77</td>
<td>87</td>
<td></td>
</tr>
</tbody>
</table>

b) Insert these numbers into an initially empty hash table with collisions resolved by chaining. Show the populated hash table. Assume the array that stores this hash table has 10 cells. [8 pts]

![Diagram of populated hash table with chaining]

- Slot 0: Empty
- Slot 1: 71 → 31
- Slot 2: Empty
- Slot 3: 23 → 73
- Slot 4: 44
- Slot 5: Empty
- Slot 6: Empty
- Slot 7: 97 → 77 → 87
- Slot 8: Empty
- Slot 9: Empty