1. Here’s an implementation of Merge sort:

```java
/** Sorts a[start..end] using mergesort. Pre: 0 <= start <= end < a.length */
public void mergeSort(int[] a, int start, int end) {
    if (end-start <= 1) {
        return;
    }
    int mid = (start+end)/2;
    mergeSort(a, start, mid);
    mergeSort(a, mid, end);
    merge(a, start, mid, end);
}
```

It makes use of the helper method `merge` that implements the following spec and runs in \(O(n)\) time.

```java
/** Merges the two sorted subarrays a[start..mid] and a[mid..end] into a
 * sorted array a[start..end] Pre: 0 <= start <= mid <= end < a.length */
public void merge(int[] a, start, mid, end);
```

(a) Let \(n = end - start\). Give the recurrence relation that describes the runtime of the `mergeSort` method:

\[
T(0) = \\
T(n) = 
\]

(b) What is the asymptotic runtime of `mergeSort`?

2. Circle T or F to indicate whether the statement is true or false.

(a) T / F The partition step of QuickSort is the “divide” phase of divide-and-conquer, whereas the merge step of MergeSort is the “conquer” phase.

(b) T / F Finding an element in a binary tree is worst-case \(O(n)\).

(c) T / F Implementing the Set ADT with a linked list would make insertion more efficient than using an array.
(d) T / F A hash table with a large load factor is more time-efficient but less space-efficient than one with a small load factor.

3. (1 pt) Which of the following could be the result of a call of the partition method in QuickSort?

(a) [ 2, 5, 2, 4, 1 ]
(b) [ 6, 2, 7, 8, 9 ]
(c) [ 6, 7, 2, 3, 4 ]
(d) [ 7, 9, 3, 4, 5 ]

4. Consider the following Binary Search Tree:

```
10
  /  \
 6   14
  /  \
 5   11
      /  \
      18  \
          /  \
         13  20
```

(a) Insert 19 using standard BST insert and draw it into the tree above.
(b) Write the sequence of necessary rotations to rebalance the tree, using “direction(value)” to denote a rotation on a node with that value. For example, left(10) indicates a left rotation on the node with value 10.

5. Consider the following three algorithms.

```
Alg1(n):
  for a = 0..n:
    for b = 0..n:
      print a + b

Alg2(n):
  for a = 0..600:
    for b = 0..(n/2):
      print a + b

Alg3(n):
  for a = 0..n:
    for b = a..n:
      print a + b
```

For each algorithm, fill the table below to indicate the number of times the algorithm prints a value and the Big-O runtime class, both in terms of n.

<table>
<thead>
<tr>
<th></th>
<th>Alg1</th>
<th>Alg2</th>
<th>Alg3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Items Printed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Big-O Runtime Class</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>