1. Shown below is the code for the bubble sort consisting of two recursive methods that replace the two nested loops that would be used in its iterative counterpart:

```
void bubbleSort(int array[]) {
  sort(array, 0);
}

void sort(int[] array, int i) {
  if (i < array.length - 1) {
     bubble(array, i, array.length - 1);
     sort(array, i + 1);
  }
}

void bubble(int[] array, int i, int j) {
  if (j <= i)
    return;
  if (array[j] < array[j - 1]) {
    int temp = array[j];
    array[j] = array[j - 1];
    array[j - 1] = temp;
  }
  bubble(array, i, j - 1);
}
```

   Draw the recursion tree for `bubbleSort` when it is called for an array of length 4 with data that represents the worst case. Show the activations of `bubbleSort`, `sort` and `bubble` in the tree. Explain how the recursion tree would be different in the best case.

2. Refer back to the recursion tree you provided in the previous problem. Determine a formula that counts the numbers of nodes in that tree. What is Big-Θ for execution time? Determine a formula that expresses the height of the tree. What is the Big-Θ for memory?

3. Provide a Java class named `SortedPriorityQueue` that implements a priority queue using a Java array of type `int`. The constructor of the class should be passed the size of the queue. Each time the `add` method is called, the value passed to it should be inserted into the array to ensure that the array remains in sorted order. If the array is full when `add` is called, a `RuntimeException` should be thrown. If the array is empty when `remove` is called, a `RuntimeException` should also be thrown Make the implementation as efficient as possible.

4. Consider the following sorting algorithm that uses the class you wrote in the previous problem:

```
void sort(int[] array)
{
  SortedPriorityQueue queue = new SortedPriorityQueue(100);
  for (int i = 0; i < array.length; i++)
    queue.add(array[i]);
  for (int i = 0; i < array.length; i++)
    array[i] = queue.remove();
```

```
}
```

Analyze its execution time efficiency in the worst case. In your analysis you may ignore the possibility that the array may overflow. Indicate whether this implementation is more or less efficient than the one that uses the Java priority queue.