1. The algorithm I have chosen is mergeSort. It is a stable sort because in the case that the terms are equal, it places the element that was already on the left in first, so it will continue to be on the left. The code snippet is here:

**if** (cIndex == c.length || c[cIndex] >= b[bIndex]){

temp[counter] = b[bIndex];

bIndex++;

counter++;

}

**else**{

temp[counter]=c[cIndex];

cIndex++;

counter++;

}

The comparison operation is a greater than or equal to check, not a strictly greater than check. Array b is the left half that is being merged, so it will enter first.

2.

The best case time complexity of this algorithm is O(NlogN). The way the algorithm works is that it breaks the array into smaller and smaller pieces until they are but sections of 1. Then it iterates through each of them and combines them. This is the merge function. The function runs in linear time, as it essentially iterates through the entire array once. However this function is called LogN times. This is because the size of the merged array doubles after each merge, so the number of times that you need to double a size of 1 to reach a size of N is logN.

This is the merge function:

**public** **int**[] Merge(**int**[] b, **int**[] c){

**int**[] temp = **new** **int**[b.length + c.length];

**int** counter = 0;

**int** bIndex = 0;

**int** cIndex = 0;

**while**(counter < b.length+c.length){

**if** (cIndex == c.length || c[cIndex] >= b[bIndex]){

temp[counter] = b[bIndex];

bIndex++;

counter++;

}

**else**{

temp[counter]=c[cIndex];

cIndex++;

counter++;

}

}

**return** temp;

}

This runs in linear time, as it is just going to loop once through each element.

3.

The worst case time complexity of mergesort is also NlogN. This is because the best and worst cases operate nearly identically, in that they will split the array into a base case of an array of length 1 and then merge them all until you get the original size back. The number of comparisons in the best and worst case is almost the same.