Problem 1

1. The algorithm I used is Merge Sort. This satisfies the ordering requirement because merge sort is a divide and conquer algorithm, which takes O(nlogn) time. This takes O(nlogn) time because it first has to divide the array into two arrays of halves, and then do it recursively. This will take log n time, and then at the end, it needs to place these values back into the original array, with n elements. This together is O(nlogn).

It is stable because when it compares the array elements of the left and right arrays, it checks if it is less than or equal to, and then will place the elements in that order. It will still place them one by one in their location.

Code:

**while**(i <=nL && j <=nR){

**if**(left[i] < right[j]){

a[k] = left[i];

i++;

k++;

}

**else**{

a[k]= right[j];

j++;

k++;

}

}

**while**(i <= nL){

a[k] = left[i];

i++;

k++;

}

**while**(j <= nR){

a[k] = right[j];

j++;

k++;

}

}

2. The best case time complexity of this algorithm is also O(nlogn), because no matter how the array values are, it will divide the array in half and then work on those halves. Then it will have to place these back into the original array. If the array elements were already sorted in the original array, it will still divide the arrays into halves, and then place them back in their original locations. It will have to make less comparisons, but will still have to divide the structure and then place all of it back into the original.

Code:

This divides it

**int**[]left = **new** **int**[m];

**int**[]right = **new** **int**[n-m];

**for**(**int** i = 0; i < m; i++){

left[i] = a[i];

}

**for**(**int** j = m; j < n; j++){

right[j-m] = a[j];

}

This places it back

**void** merge(**int**[] left, **int**[] right, **int**[]a){

**int** nL = left.length;

**int** nR = right.length;

**int** i = 0;

**int** j = 0;

**int** k = 0;

**while**(i <=nL && j <=nR){

**if**(left[i] < right[j]){

a[k] = left[i];

i++;

k++;

}

**else**{

a[k]= right[j];

j++;

k++;

}

}

**while**(i <= nL){

a[k] = left[i];

i++;

k++;

}

**while**(j <= nR){

a[k] = right[j];

j++;

k++;

}

}

3. The worst case time complexity of this algorithm is also O(nlogn). This is because even if the array elements in the original array are in descending order, no matter what the values are in the array, to sort it, it will divide it into halves and then place the elements back in their correct order using if statements and comparisons. This will happen every time, and is why it will always be O(nlogn) for every case.

**void** ExamSort(**int**[]a, **int** size){

**int** n = a.length;

**int** m = n/2;

**int**[]left = **new** **int**[m];

**int**[]right = **new** **int**[n-m];

**for**(**int** i = 0; i < m; i++){

left[i] = a[i];

}

**for**(**int** j = m; j < n; j++){

right[j-m] = a[j];

}

ExamSort(left,m);

ExamSort(right,n-m);

merge(left,right,a);

}