1. I utilized MergeSort in order to answer this problem. This satisfies the relative ordering requirement, because when merging, we have two indexes that go through the left half and right half of the array (i and j), and in cases where both might point to the same value, we always copy from the left half before the right half. When this happens, it is because the copy in my else statement (on line 30) is the one which happens, (none of the if, or else-ifs are satisfied). Furthermore, since we are always copying from lo to mid for the left half, and mid+1 to hi for the right half, (always iterating by 1), there will never be a case where we merge things out of order.

23 **public** **static** **void** merge(**int**[] a, **int**[] aux, **int** lo, **int** mid, **int** hi){

24 **for**(**int** k = lo; k <= hi; k++) aux[k] = a[k];

25 **int** i = lo, j = mid+1;

26 **for**(**int** k = lo; k <= hi; k++) {

27 **if**(i>mid) a[k] = aux[j++];

28 **else** **if**(j>hi) a[k] = aux[i++];

29 **else** **if**(aux[j]<aux[i]) a[k] = aux[j++];

30 **else** a[k] =aux[i++];

31 }

32 }

2. The time complexity of ExamSort will not vary depending on input, it will always perform the same amount of operations. Therefore the best case for ExamSort can be given simply by analyzing the algorithm. ExamSort takes a divide and conquer approach to sorting, by recursively calling ExamSort on the left and right halves of the array then merging both sorted halves. These recursive calls give a Complexity of O(logN). The merge function has complexity O(N), since it has a for loop which runs related to N times (from lo to hi).

Combining both parts give a complexity of **O(NlogN)**, since the merge function will be called logN times and has complexity N.

15 **public** **static** **void** ExamSort(**int**[] a, **int**[] aux, **int** lo, **int** hi){

16 **if**(lo>= hi) **return**;

17 **int** mid = (lo+hi)/2;

18 *ExamSort*(a, aux, lo, mid);

19 *ExamSort*(a, aux, mid+1, hi);

20 *merge*(a,aux,lo,mid,hi);

21 }

22

23 **public** **static** **void** merge(**int**[] a, **int**[] aux, **int** lo, **int** mid, **int** hi){

24 **for**(**int** k = lo; k <= hi; k++) aux[k] = a[k];

25 **int** i = lo, j = mid+1;

26 **for**(**int** k = lo; k <= hi; k++) {

27 **if**(i>mid) a[k] = aux[j++];

28 **else** **if**(j>hi) a[k] = aux[i++];

29 **else** **if**(aux[j]<aux[i]) a[k] = aux[j++];

30 **else** a[k] =aux[i++];

31 }

32 }

3. The time complexity of MergeSort in the worst-case will be the same as the best case, and as such can be found the same way. ExamSort takes a divide and conquer approach to sorting, by recursively calling ExamSort on the left and right halves of the array then merging both sorted halves. These recursive calls give a Complexity of O(logN). The merge function has complexity O(N), since it has a for loop which runs related to N times (from lo to hi).

Combining both parts give a complexity of **O(NlogN)**, since the merge function will be called logN times and has complexity N.

15 **public** **static** **void** ExamSort(**int**[] a, **int**[] aux, **int** lo, **int** hi){

16 **if**(lo>= hi) **return**;

17 **int** mid = (lo+hi)/2;

18 *ExamSort*(a, aux, lo, mid);

19 *ExamSort*(a, aux, mid+1, hi);

20 *merge*(a,aux,lo,mid,hi);

21 }

22

23 **public** **static** **void** merge(**int**[] a, **int**[] aux, **int** lo, **int** mid, **int** hi){

24 **for**(**int** k = lo; k <= hi; k++) aux[k] = a[k];

25 **int** i = lo, j = mid+1;

26 **for**(**int** k = lo; k <= hi; k++) {

27 **if**(i>mid) a[k] = aux[j++];

28 **else** **if**(j>hi) a[k] = aux[i++];

29 **else** **if**(aux[j]<aux[i]) a[k] = aux[j++];

30 **else** a[k] =aux[i++];

31 }

32 }