**public** **static** **void** ExamSort(**int**[] a, **int** size)

{

**int**[] temp = **new** **int**[a.length];

*mergeSort*(a, temp, 0, a.length-1);

}

**private** **static** **void** mergeSort(**int**[] a, **int**[] aux, **int** lo, **int** hi)

{

**if**(hi<=lo) **return**;

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

*mergeSort*(a,aux,lo,mid);

*mergeSort*(a,aux,mid+1,hi);

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

}

**private** **static** **void** merge(**int**[] a, **int**[] aux, **int** lo, **int** mid, **int** hi)

{

**if**(a[mid]<a[mid+1]) **return**;

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

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

{

aux[k]=a[k];

}

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

{

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

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

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

**else** a[k]=aux[j++];

}

}

1. I used a mergeSort as it runs in O(NlgN) (due to the fact that it halves the array and sorts each half, hence the logarithmic nature), and because it is a stable sort since it preserves order of elements with equal keys.

2. The best and worst case of mergeSort are O(NlgN). Calling mergeSort on left and right halves is always executed and will always execute in O(lgN) time. Calling merge on the two subarrays always requires N array accesses. Therefore regardless of the array, there will be lgN number of ‘splits’ and at each one, there will be N array accesses.

3. As mentioned above, O(NlgN) is the time complexity regardless of best or worst case.