Sri Jay Adarsh Gogineni

Ruid 171003252

Problem 1)

1. The algorithm of my choice was MergeSort. Merge sort is a stable algorithm because of the fact that when it divides up the array into one element arrays and merges them back together, we can check for the relative ordering of these two numbers. If the number in the left array is EQUAL to the number in the right array, we always choose the left array to be copied back into the original array FIRST. This allows for the correct ordering of the two equal numbers to be saved in the sorted array. This code shows this:

**while**(i < left.length && j < right.length){

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

a[k++] = left[i];

i++;

}

**else**{

a[k++] = right[j];

j++;

}

}

As you can see, the if the left array element is less than or equal to the right element, the left gets copied in first.

2.The best-cast time complexity of this algorithm is O(NlogN). This is because even if the original array is sorted, the merge sort first calls the splitting, which takes logN time and then compares the split arrays against each other which takes O(N) time.

*ExamSort*(left,left.length);

*ExamSort*(right,right.length);

*Merge*(a,left,right);

3. The worst-case time complexity of this algorithm is O(NlogN). This is also as explained above. The unorderedness of the array does not affect the time complexity since the merge sort will always call the splitting first and then the comparing of the arrays.

*ExamSort*(left,left.length);

*ExamSort*(right,right.length);

*Merge*(a,left,right);