Matthew Jackson Problem 1 11/2/2018

1. For this sort, I used merge sort since it runs at a time complexity of O(NlogN) in all cases and is stable. Merge sort divides an array into halves in the order that the array is inputted until it is no longer possible to divide the array into halves. When the halves are merged together, while, for and if-else statements are used to retain the order of the array. Since it retains the order that the array is inputted, merge sort is stable.

Dividing: int mid = arrSize / 2;

int[] L = new int[mid];

int[] R = new int[arrSize - mid];

for (int i = 0; i < mid; i++){ //For statement that retains order.

L[i] = arr[i];

}

for (int i = mid; i < arrSize; i++){ //Another for statement that retains order.

R[i - mid] = arr[i];

}

Merging: int i = 0, j = 0, k = 0;

while (i < left && j < right){ //while statement that retains order.

if (L[i] < R[j]){ //If-else statement used to retain order.

arr[k++] = L[i++];

} else {

arr[k++] = R[j++];

}

}

while (i < left){ //Another while statement that retains order.

arr[k++] = L[i++];

}

while (j < right){ //Another while statement that retains order.

arr[k++] = R[j++];

}

}

2. The best case time complexity of merge sort is O(NlogN) because it will have to divide the array into halves which takes logN time. Merging the array takes N time since it is dealing with an array of N size. Altogether this creates a time complexity of O(NlogN).

3. The worst case time complexity of merge sort is O(NlogN) because the order of the inputted array does not affect how long it takes merge sort to sort an array. Since the array will always be divided (which takes logN time) and merged back together (takes N time) regardless of the order of the inputted array, the time complexity of mergeSort is the same in the best case and worst case.