2B)

Merge sort has a complexity of O(log n). This is because the code is taking up all the elements and divide until there are one element left then it will start by sorting two, fours, and so on until it completes the size of the array. However, it will loop this until the code is sorted into two where the first half to midpoint is sorted and the second half of midpoint is also sorted. Then it will do one final sort where it sorts the two halves that were sorted. Therefore overall the complexity is O(NlogN) where the first n is the sorted after the halves are sorted and the log N is the recursion factor of the sort.

```
//sort the front and back of the midpoints
while(i <= j && j <= k) {
        if(a[i] > a[j]) {
            int temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            i++;
            j++;
        }
        else i++;
}

//merge two midpoints
for (int b = 0; b < j; b++){
    for(int c = j-1; c<a.length; c++){
        if (a[c]<a[b]){
            int temp = a[c];
            for(int v = c; v > b-1; v--){
                if(v-1 >= 0) {
                    a[v] = a[v-1];
                }
            }
            a[b] = temp;
            b++;
        }
    }
}
```

This code is stable because of recursion. This means the original sort before being sorted again will still have its index spot saved therefore after the recursion, the same number that was repeated in the array will still be saved in the final array where it will be first in the sorted array as well. The code below shows the recursion factor.

```
public static void sort(int[]a, int left, int right) {
    if(left < right) {
        int mid = (left + right)/2;
        sort(a, left, mid);
        sort(a, mid + 1, right);
        mergeSort(a, left, mid, right);
        }
}
```