Part A: Time Complexities Analysis

Merge sort:  
 Merge sort uses thr sortAux method recursively divides the map into 2 sublists until the base case is reached (when left >= right). This method works (log n) times where the n is size of the map and each call it calls in all cases(best, average, worst). The merge method merges two sorted sublists of the auxiliary list into one sorted sublist. It iterates through the two sublists and compares the elements to determine their order and after one of them finish adds remaining elements of the remain one. The merging process have some loops that working (n) times and using get and put methods of the hashmap that have time complexity O(1)\* and set method of arraylist that is also O(1) thats why merge methos has time complexity of O(N) in all cases.

Since the merge method is called at each level of recursion in the sortAux method, the total time complexity is (log n) times O(N) therefore O(n log n).  
In conclusion,

Best case: O(N log N)

Average case: O(N log N)

Worst case: O(N log N)  
Which is N is size of the map

\*Source: https://www.tutorialspoint.com/Difference-between-TreeMap-HashMap-and-LinkedHashMap-in-Java#:~:text=LinkedHashMap%20has%20complexity%20of%20O,key%2Dvalue%20pairs%20are%20inserted.

Selection sort:  
In the sortAux method the outer loop runs n - 1 times, where n is the size of map. This loop iterates through each element of the list and the inner loop also runs n - i times, which i is work count of outer loop. This loop finds the minimum element in the unsorted portion of the list.

The comparison in the inner loop’s if (originalMap.getCount(aux.get(j)) < originalMap.getCount(aux.get(minIndex))) takes constant time which we discussed in merge sort’s analysis.

The time complexity of selection sort is O(n^2) beacuse nested loops that works n times and constant time for each iteration, where n is the number of elements in the list.   
In conclusion,

Best case: O(N^2)

Average case: O(N^2)

Worst case: O(N^2)  
Which is N is size of the map

Insertion sort:  
 In the insertion sort and other remainig algorithms the time complexsity of the cases not the same that’s why I will evaluate separately.  
Best-case time complexity: O(n)

In the best-case scenario, when the input list is already sorted, the algorithm will perform a single comparison for each element in the auxiliary list. As a result, it will skip the inner while loop and complete the sorting process in linear time, resulting in a complexity of O(n) which is very fast.

Average-case time complexity: O(n^2)

In the average-case scenario, the algorithm will require comparisons and potential swaps for each element in the auxiliary list. For each element, it will compare it with all the preceding elements until it finds its correct position. On average case hastime complexity of O(n^2).

Worst-case time complexity: O(n^2)

In the worst-case scenario, when the input list is sorted in reverse order, the algorithm will perform a comparison and a potential swap for each element in the auxiliary list. This will result in nested loops that iterate over the entire auxiliary list, leading to a time complexity of O(n^2).

Bubble sort:  
 Best-case time complexity: O(N)

In the best-case scenario, the input list is already sorted. When the input list is sorted, the algorithm only performs a single pass over the list to confirm that it is *sorted*. In this case, the inner loop will run for (n - i - 1) iterations in each iteration of the outer loop(1 for best case), where n is the size of the list and i is the iteration number of the outer loop. As a result, the time complexity of the algorithm in the best case is O(n), where n is the number of elements in the list.

Average-case time complexity: O(N^2)

In the average-case scenario, the input list is randomly ordered. In each iteration of the outer loop, the inner loop compares adjacent elements and swaps them if they are in the wrong order. On average, the inner loop will run approximately n/2 times for each iteration of the outer loop. Therefore, the average number of comparisons and swaps in each iteration of the outer loop is proportional to n/2. Since there are n-1 iterations in the outer loop, the total number of comparisons and swaps is approximately (n/2) \* (n-1). Thus, the average-case time complexity of the algorithm is O(n^2), where n is the number of elements in the list.

Worst-case time complexity: O(N^2)

In the worst-case scenario, the input list is sorted in descending order. In this case, the algorithm will need to perform the maximum number of comparisons and swaps. The inner loop will run (n - i - 1) times in each iteration of the outer loop, resulting in a total of (n-1) + (n-2) + ... + 1 = (n^2 - n) / 2 comparisons and swaps. Therefore, the worst-case time complexity of the algorithm is O(n^2), where n is the number of elements in the list.

Quick sort:

Best-case time complexity: O(n log n)

The best-case scenario occurs when the pivot element chosen during partitioning divides the array into two equal-sized subarrays. In this case, each recursive call will operate on a subarray of approximately half the size of the previous subarray. Therefore, the partitioning process will made (log n) times, and since it needs to be performed for each level of recursion, the total time complexity becomes O(n log n).

Average-case time complexity: O(n log n)

The average-case scenario assumes that the input elements are randomly distributed. The quickSort algorithm divides the array into subarrays based on the pivot element, and each partitioning step takes linear time. On average, the pivot will divide the array into two subarrays of approximately equal size in average. Therefore, the partitioning process will take O(n) time. Since the algorithm recursively operates on both subarrays, the average time complexity can be approximated to O(n log n).

Worst-case time complexity: O(n^2)

The worst-case scenario occurs when the input array is already normal or reverse sorted. In this case, if the pivot element is consistently chosen as the smallest or largest element, one of the subarrays will have a size of n-1, and other array contains no elemnt. This leads to highly unbalanced partitions and results in a worst-case time complexity of O(n^2).

Part B: Measured Running Times

This timesmeasured with runTimes.java file in homework7 directory, for each case of each algorithm running 100 times wtih same unsorted array and take average of these for more stabilezed results:

SORTING WITH MERGE SORT...

Worst case: 0,028048 ms

Avarage case: 0,014088 ms

Best case: 0,008449 ms

SORTING WITH SELECTION SORT...

Worst case: 0,003452 ms

Avarage case: 0,002778 ms

Best case: 0,001584 ms

SORTING WITH INSERTION SORT...

Worst case: 0,002802 ms

Avarage case: 0,002353 ms

Best case: 0,001377 ms

SORTING WITH BUBBLE SORT...

Worst case: 0,003642 ms

Avarage case: 0,002612 ms

Best case: 0,002385 ms

SORTING WITH QUICK SORT...

Worst case: 0,004950 ms

Avarage case: 0,003851 ms

Best case: 0,001581 ms

Part C: Use Cases and Best Algorithms

When we know our inputs are mostly sorted but we don’t sure completly sorted or some elements sorted or not, in this cases therotically we have 2 algorithms that perfrorms O(N) time complexsity which is incredibly fast for a sort algorithm, one of them is bubble sort and insertion. While our inputs are very large that constant time things becomes not imported, we should use this two algorithm but when the our inouts small cont time things become more important and we can observe that with measured times and this case selection and insertion sort have pretty good results and we should use those.  
  
When we don’t know the input’s case or inputs randomly ordered we can trust the average cases and wile we have large inputs theoric results show us quick sort and merge sort best in average case if we have enough space we can use merge sort but when we have limited space merge sort connot be executable and we should quick sort or if we have more limited space we have to use other akgorithms which is perform O(N^2) time complexsity in average case beacuse space complexity of merge sort algorithm directly proportional to the size of the map and in quick sort it is (log n) which is n is size of the map and others have constant space complexity. When we have smaller inputs, we can look at the running times we measured above, we see that selection, insertion, bubble and quick (best case of quick sort which is best for this case)give good results.  
And finally when we know the inputs are mostly reverse ordered which is worst case of algorithms and performs O(N^2) except merge sort that’s why in large inputs we can say obviously we should use merge sort for faster results(if we have enough space) but in small inputs still we should look measured times and it shows us insertion selection and bubble gives not bad results to us.

Part D: Same Count Case

Merge sort:

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu  
Merge sort perform same order with input beacuse if two count value is same, algorithm adds left one first which is comes before in input and other will be added after left one is bigger than right one that’s why it’s same ordered

Selection sort:

metin, ekran görüntüsü, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu  
Selection picks first occurrence of smallest element of unsorted part and replace it with first elemnt of unsorted part that’s why for smallest count of whole map input orfer protected but for other element it isn’t guaranteed beacuse of the previous element replacings may have changed the order examply if counts and lettters 1e, 2c,2f,1d when sorting 1d will replaces with 2c and 2c will become after 2f and order couldn’t be protect.

Insertion sort:  
ekran görüntüsü, metin, yazılım, multimedya yazılımı içeren bir resim

Açıklama otomatik olarak oluşturuldu  
Insertion sort performs same order with input beacuse it start at the end move element until it smaller than the previous element, therefore when it finds same value of count it stops and input order have protected.

Bubble sort:

metin, ekran görüntüsü, yazılım, multimedya yazılımı içeren bir resim

Açıklama otomatik olarak oluşturuldu  
Buuble sort performs same order with input either beacuse it starts at the first elemnt and moves it until it smaller tha next element and when it finds bigger element it takes that and continue the same operation, therefore when it finds same count value it will stop and input order have protected.

Quick sort:  
Similiarly with selection sort this algorithm makes replacing and the first occurrence may be replaced with other next elements and when this accurs order is autamitcally destroy.

metin, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu