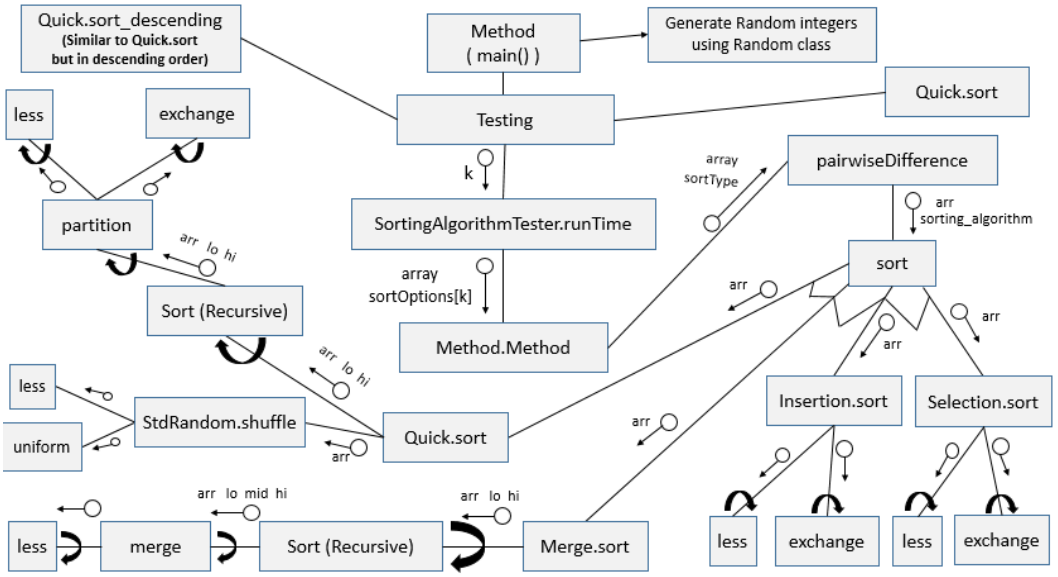
Basme Zantout **Report**

➔ Problem Statement and Code Design

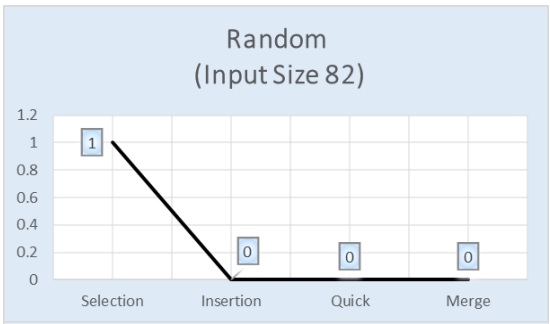


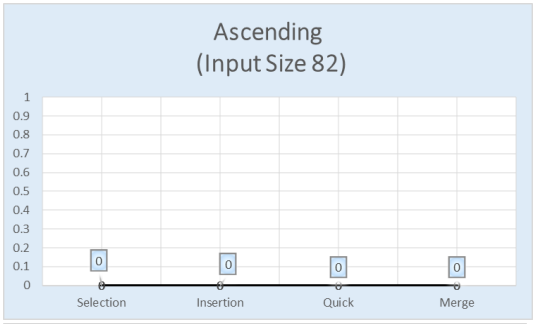
➔ Implementation, Functionality

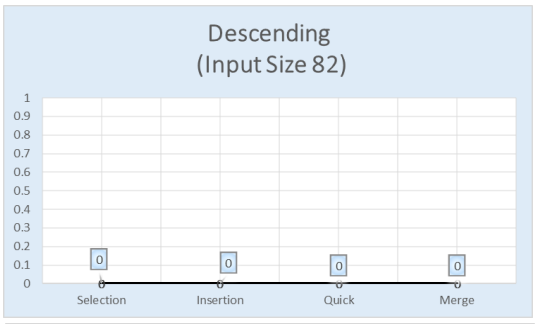
First, the main class called **"Method"** proposes 5 different Input sizes to be tested using different sorting algorithms. So, to do so we read an input seed value and instantiate the *static Integer array* in SortingAlgorithmTester class with 5 different sizes and with Random integers (using the **Random** class). Then, for each input size, we call the **Testing** method which calls the **SortingAlgorithmTester.runTime** method 3 times for each sorting algorithm. In other words, for each input size we call the **Testing** method which passes a parameter to the **runTime** method indicating what sorting algorithm will be used, and it does this 3 times for each sorting algorithm: **1-** for Randomly-ordered array **2-** For array with ascending order (I used **Quick.sort** for this) **3-** For array with descending order (I used the reverse of Quick.sort which is **Quick.sort_descending** for this). Afterwards, the **runtime** method calls the **Method** method which takes the array as parameter as well as the sorting type that will be used to in order to test its ***run time***, and it calls the **pairWisedifference** method which takes the array and sorting parameters from the **Method** method and calculates the smallest pairwise difference after sorting the array using the indicated sorting type in the parameter. For the **pairWisedifference** method to choose the sorting algorithm easier, it calls a method named **sort** in the main class and this method will use a switch-case statement to choose and call the right sorting algorithm to be used. Thus, it will call one of the following sorting algorithms: **Quick.sort** or **Merge.sort** or **Selection.sort** or **Insertion.sort** (The standard sorting algorithm is used in each class relative to its type).

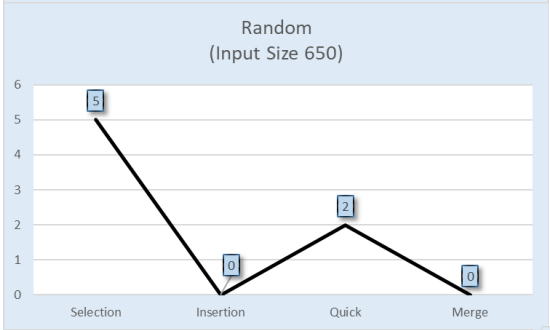
➔ Testing

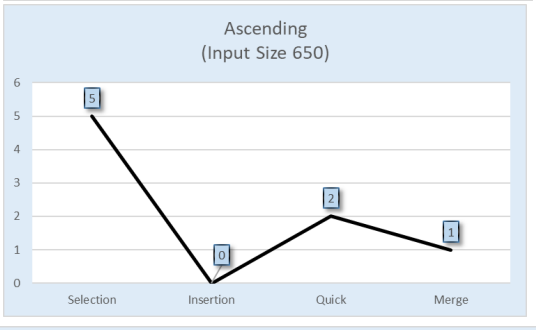
Using the "SortingAlgorithmTester.java" class we have calculated the approximate run time of different sorting algorithms on arrays of different sizes and orders, the results were as follow (Note: the left vertical axis shows the run time):

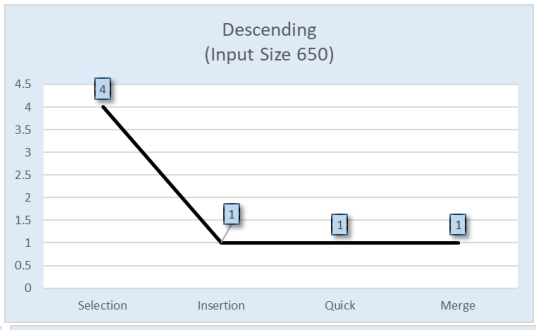


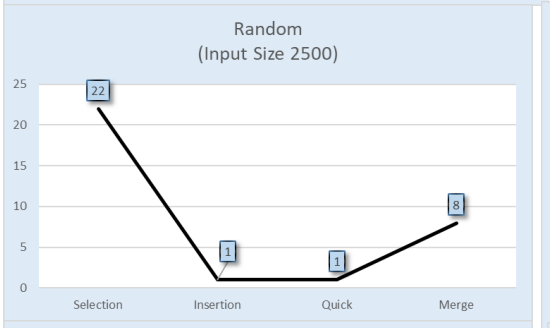


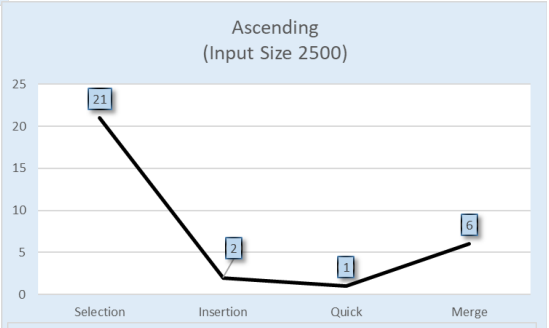


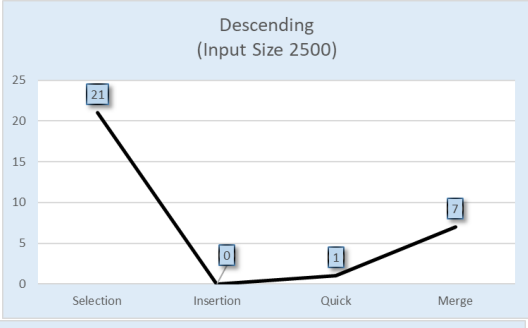


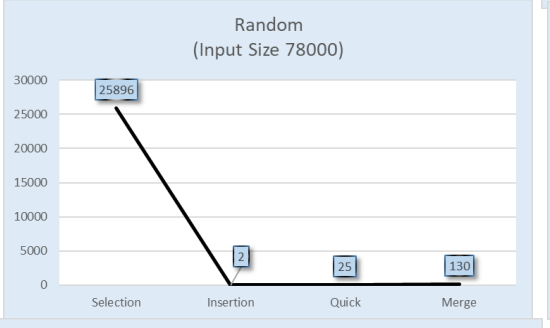


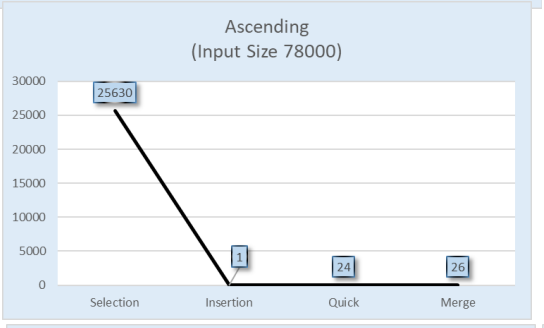


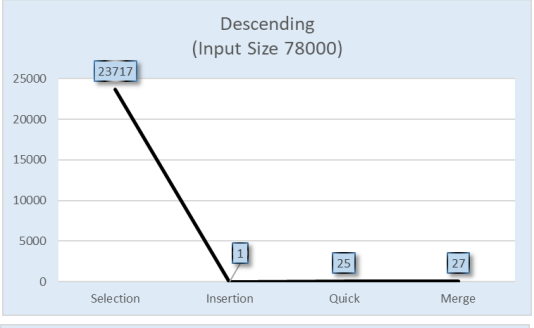


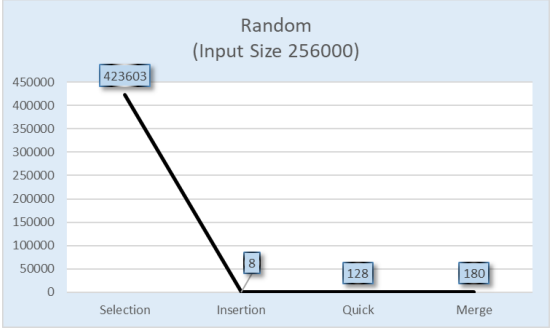


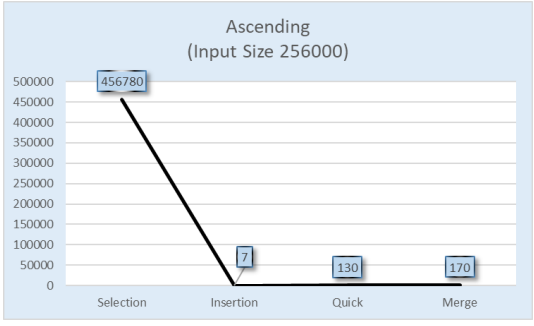


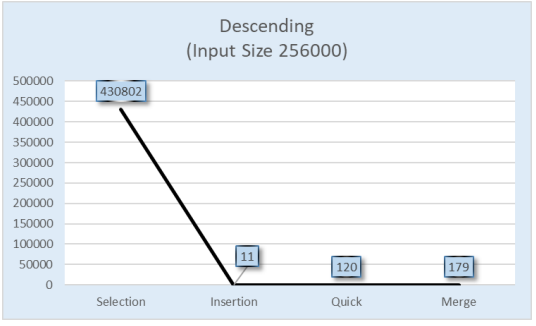










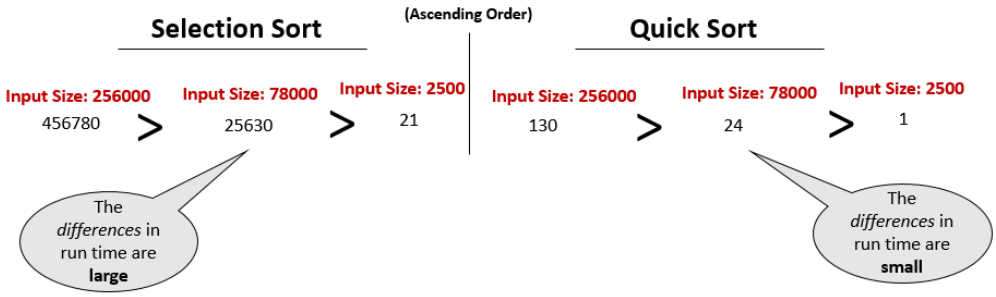


Sorting Algorithm: The Sorting Algorithms tested are: **Selection, Insertion, Quick,** and **Merge** sorts. As we can observe from the charts above, the **Selection Sort** algorithm proved to be the least efficient among all sort algorithms tested, from the least input size tested to the greatest one, Selection sort always took the greatest run time. Although on average **Insertion Sort** yields relatively the lowest run time among all as indicated in the charts, when observing the code during compilation, **Insertion Sort** took much more time than **Merge** and **Quick Sorts** but a little less time than **Selection Sort**. In other words, while it took several *minutes* to get the results when using **Insertion** and **Selection Sorts, Merge** and **Quick Sorts** were done in a matter of *seconds*. Hence in general, the code ran the fastest in **Quick** and **Merge Sorts** and the slowest in **Selection** and **Insertion Sorts**.

Array Order: The array orders tested are: **Random Order, Ascending Order,** and **Descending order.** When doing the comparisons of run time based on the order of the array, we can deduce that the **Randomly-ordered** arrays take a little more time than arrays of **Ascending** and **Descending orders**, for example:

However, the difference is so slight that it does not have a huge effect on the run time and can therefore be considered negligible.

Input Size: The input sizes tested are: **82, 650, 2500, 78000,** and **256000**. Reasonably, as the input size of the array increases the run time increases too. However, in small input sizes like **82, 650,** and **2500** the run time occurs in seconds or even milliseconds in all sorting algorithms and orders, while in larger input sizes like **78000** and **256000** it took many minutes to sort the array when we applied Selection and Insertion Sorts. So, we can conclude that when the **input size is relatively small** the type of sorting algorithm chosen is insignificant to the run time, but when the **input size gets larger** it would be more efficient to use Quick and Merge Sort algorithms. For example:



➔ Final Assessment

One of the trouble points that I have faced in this assignment is dealing with large input sizes. In practices, we usually deal with relatively small input sizes that work very fast and efficiently in many codes and in almost all sorting algorithms, but in this assignment we are testing our code on large input sizes, which requires putting into consideration the run time as well as the space that will be used during compiling our code. Therefore, the testing part was the most challenging part for me in this assignment. However, what I liked about this assignment is that it provided us with hands-on activity on somehow real life examples. Therefore, we learned from this assignment about the real challenges of memory space and data size problems that we might face in the future.