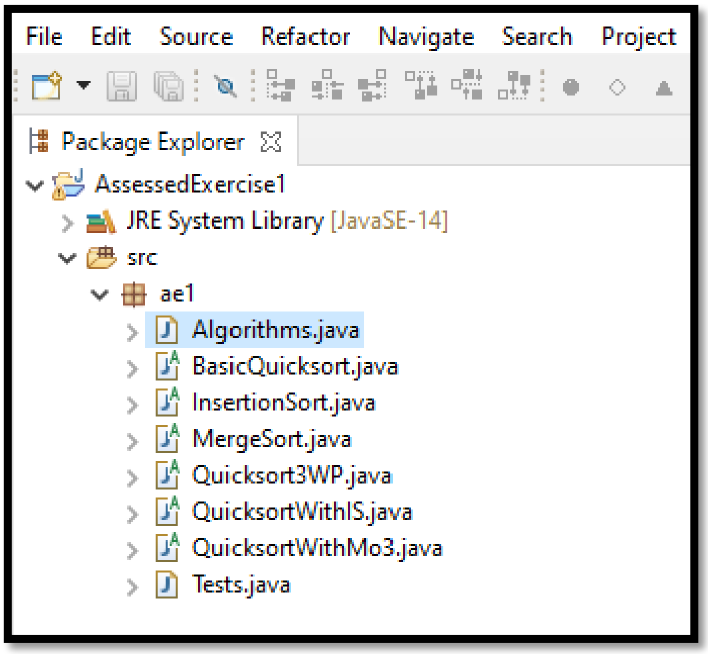
ANALYSIS OF SORTING ALGORITHMS

How to run the code

You will be provided with a .zip file named ***comparison_sorting_algorithms*** containing all the necessary material to run the program (including test cases). You will have to unzip the folder and create a workspace to store the Java project. You should then have the following hierarchy:



```
AssessedExercise1
  JRE System Library [JavaSE-14]
  src
    ae1
      Algorithms.java
      BasicQuicksort.java
      InsertionSort.java
      MergeSort.java
      Quicksort3WP.java
      QuicksortWithIS.java
      QuicksortWithMo3.java
      Tests.java
```

Algorithms.java contains the main method from which you will run the algorithms. Note that each algorithm implementation is in a separate abstract class. To use any of the algorithms you will have to make a static reference to the method in each class:

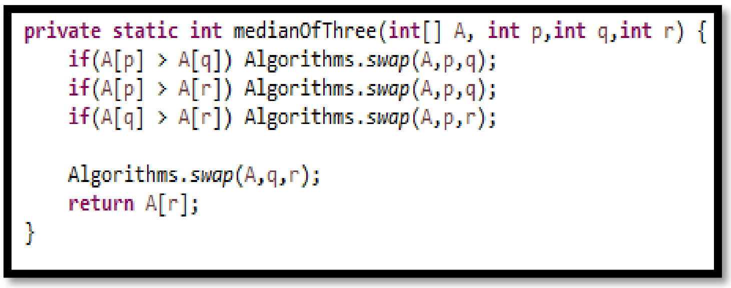
- **For the basic quicksort**: BasicQuicksort.quicksort()
- **For quicksort with insertion sort**: QuicksortWithIS. quicksortWithInsertionSort()
- **For quicksort with middle of three partition**: QuicksortWithMo3.quicksortWithMedianOfThree()
- **For quicksort with 3-way partition**: Quicksort3WP.quicksortWith3wayPartition()
- **For insertion sort**: InsertionSort.insertionSort()
- **For merge sort**: MergeSort.mergeSort()

The ***Tests.java*** file contains all the appropriate tests that must be carried out for QUESTION 2 & 3 of the Assessed Exercise. A reference to the methods required for testing will be already coded in the main() method of the Algorithms.java class.

Implementation of the Sorting Algorithms

The implementation of Quicksort and Quicksort with Insertion Sort is quite trivial. We just had to implement the pseudocode provided. However, the implementation of Quicksort with median of three partition and the 3-way partition quicksort required more thinking.

Quicksort with Median of Three Partition. In principle, this variation of the quicksort algorithm only changes the way we choose the pivot. Before, we always selected the right-most element but now, we will choose the median of three elements from the array (elements in the first, middle and last index) as our pivot. This requires an additional method in charge of computing the median of the aforementioned values and reordering these three elements from the array.



```java
private static int medianOfThree(int[] A, int p,int q,int r) {
    if(A[p] > A[q]) Algorithms.swap(A,p,q);
    if(A[p] > A[r]) Algorithms.swap(A,p,q);
    if(A[q] > A[r]) Algorithms.swap(A,p,r);

    Algorithms.swap(A,q,r);
    return A[r];
}
```

3-Way Partition Quicksort. Arguably, this has been the most complex algorithm to implement. On the surface, we split (partition) the array into three parts which are less than, greater than or equal to the pivot[2]. We then iterate through the array, and depending on the current element relation with the pivot, we do the following:

- ***If the element is equal to the pivot***: we increment the current index by 1.
- ***If the element is greater than the pivot***: we swap the current index with the greatest index and decrease this by 1.
- ***If the element is less than the pivot***: we swap the current least index and the current index and increase both the index of the least element and the current index by 1.

We then return an array containing the index less than and greater than the pivot (after the execution of the while loop). We then use this to recursively call the quicksort algorithm.

(Note that this implementation is further explained as comments in the appropriate Java class)

<u>Comparison of Algorithms – QUESTION 2</u>

For the following files: "int10.txt", "int50.txt", "int100.txt" the results are clear. All of the sorting algorithms perform very well, with a total running time of less than or equal to 0ms – that is, ***all the algorithms work well with a 'small' number of array elements***. The interesting results show up when we compare the performance of the sorting algorithms on large arrays.

Files	Quicksort	Quicksort/InsertionSort	MedianOfThree Partition	3-Way Partition
Int20k	1ms	1ms	1ms	1ms
Int500k	40ms	38ms	42ms	48ms
intBig	86ms	80ms	87ms	98ms
dutch	295ms	498ms	296ms	26ms

Figure 1 Cut-off value of 40 results

For instance, when testing the 'intBig.txt' and 'int500k' files, quicksort with the insertion sort implementation is the fastest algorithm of them all. This implies that the cut-off value of 40 works well for these two files. However, running time increases substantially when testing the 'dutch' file which means that the cut-off value chosen is inappropriate for this file.

Files	Quicksort	Quicksort/InsertionSort	MedianOfThree Partition	3-Way Partition
Int500k	39ms	**57ms**	42ms	49ms
intBig	85ms	117ms	88ms	97ms
dutch	286ms	165ms	302ms	26ms

Figure 2 Cut-off value of 1000 results

Note how the increase in the cut-off value enhances the quicksort / insertion sort algorithm for the dutch file but negatively affects the 'intBig' and 'int500k' files. With this we can conclude that there is no appropriate cut-off value for every single array. Some will benefit from smaller cut-off values while others won't: ***the cut-off value depends on the size of the array***. A value

2 – In my implementation, the pivot is the leftmost element of the array

between 10 and 150 should work fine for most arrays, but as aforementioned, there are always exceptions.

From figure 1, we can see how the 3-Way Partition works extremely well with the 'dutch' file with only 26ms of running time compared to the 295ms of the unmodified quicksort. The median of three partition seems to be giving unexpected results since it does not really outperform any of the other algorithms. There are various methods of implementing this partition. In my code, I chose to find the median of the first, middle and last index of the given array. Maybe if I were to choose the elements randomly and then derive the median, I would have gotten better results. Nevertheless, it is still a good sorting algorithm (much better than insertion sort and – to a lesser extend – merge sort).

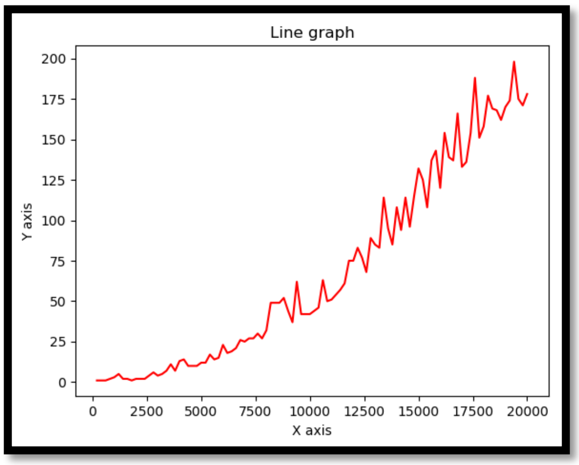
Insertion sort works fine for 'relatively small arrays' (less than 1000 elements) but it soon becomes extremely slow (*i.e.: intBig running time is of 86612ms while mergeSort is much less, 182ms*). Even though mergeSort performs much better than insertionSort, it is still quite slow compared to quicksort (as expected). These were the result:

Files	Insertion Sort	MergeSort
Int20k	30ms	2ms
Int500k	20897ms	89ms
intBig	86612ms	182ms
dutch	20874ms	82ms

Figure 3 InsertionSort and MergeSort running time results

Killer Array for Quicksort (Median-of-Three Partition) – QUESTION 3

The killer array algorithm implemented is fairly straightforward. We just generate an ordered array and then we shift the array rightward so that the pivot is the largest element of the array (excluding the last index). This causes an uneven split of the array since most elements will be smaller than the pivot, so the main advantage of the quicksort algorithm disappears giving the worst-case running time of O(n^2).



Clearly, we are dealing with a time complexity of O(n^2) as we can see from the graph. In order to get the graph and result, the size of the array was increased by 200 every time the quicksort algorithm was run and running time results were stored. We then plotted a graph using matplot library (python). The source code for this file is inside the .zip file as plotter.py.

Aside – Related to Running time comparisons

It is almost impossible to get 'perfect' results since the running time is affected by many external factors (not just the algorithm). This is why we get slightly different results for the running times every time we run the program. Nevertheless, the results provided in this document helped to determine which algorithm works best for different array implementations. You can find all the results information inside the '*results'* folder, alongside a .png of the graph.