**Lab Report**

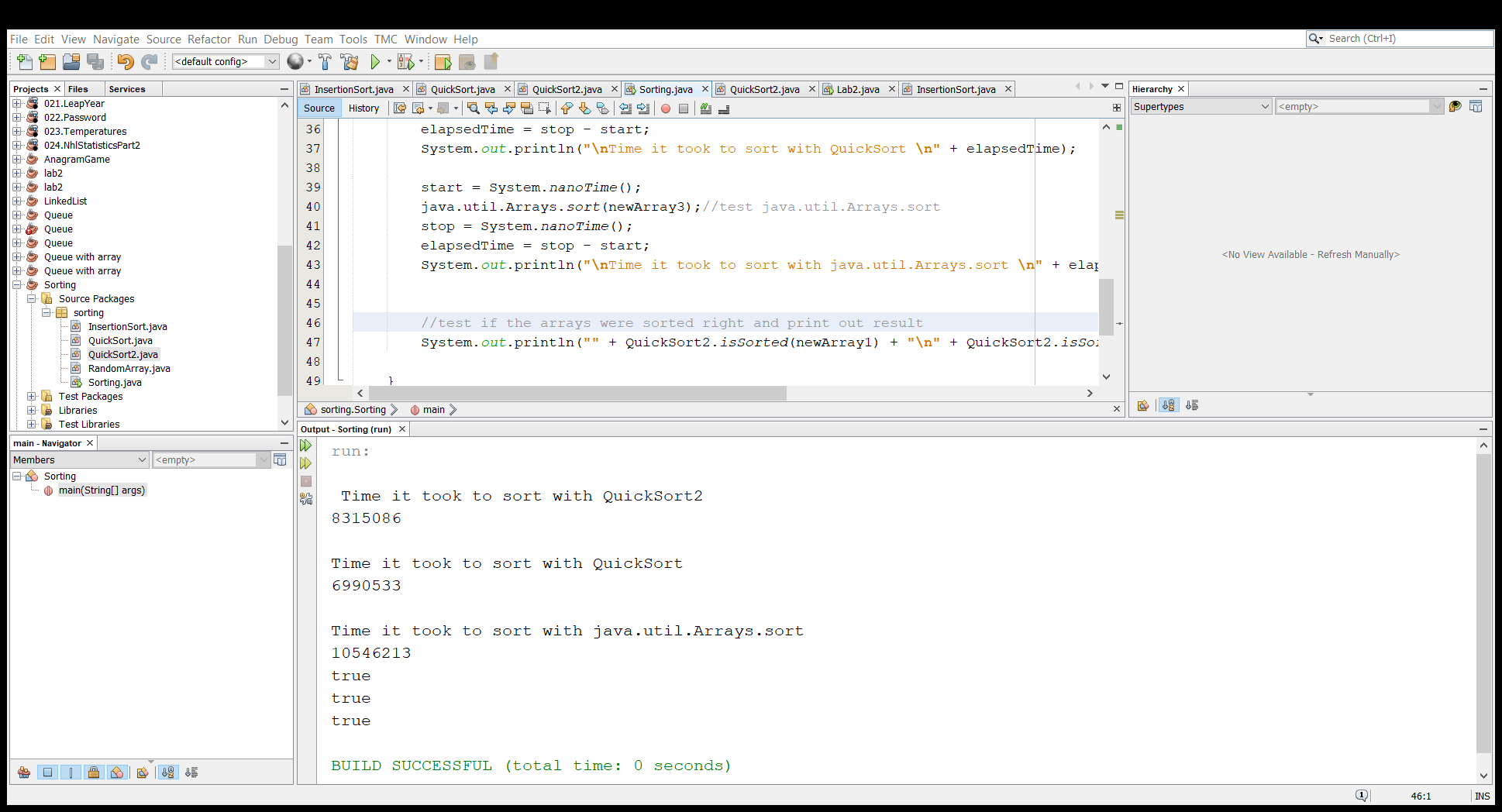
|  |  |  |  |
| --- | --- | --- | --- |
| **Lab Number:** | *2* | **Date:** | *2018-10-21* |
| **Participant:** | *Ke Zhang* | | |
| **Participant:** | *Audrone Ravdaite* | | |

**Task 1**

The code for QuickSort2

|  |
| --- |
| public class QuickSort2 {  public static void sort(Comparable[] array, int length) {  sort(array, 0, array.length - 1, length);  }  static void sort(Comparable[] array, int low, int high, int length) {  if (low < high) {  int pivot = partition(array, low, high);  int minLength = length;    if ((pivot - low - 1) < minLength) {  InsertionSort.sort(array, low, pivot + 1);  } else {sort(array, low, pivot - 1, minLength); //sort left}  }  if ((high - pivot + 1) < minLength) {  InsertionSort.sort(array, pivot + 1, high + 1);    } else {sort(array, pivot + 1, high, minLength);//sort right}    }  }  static int partition(Comparable[] array, int low, int high) {  Comparable pivot = array[high];  int i = (low - 1);  for (int j = low; j < high; j++) {  if (lessEqual(array[j], pivot)) {  i++;  exchange(array, i, j);  }  }  exchange(array, i + 1, high);  return i + 1;  }  private static boolean lessEqual(Comparable v, Comparable w) {  return (v.compareTo(w) <= 0);  }  private static void exchange(Comparable[] array, int i, int j) {  Comparable t = array[i];  array[i] = array[j];  array[j] = t;  }  public static boolean isSorted(Double[] a) {  for (int i = 0; i < a.length - 1; i++) {  if (a[i] > a[i + 1]) {  return false;  }  }  return true;  }  } |
| **Listing 1:** Code for QuickSort |

Screenshot of the test of whether the arrays were sorted correctly



**c.** Discuss in a short paragraph the possible advantages and disadvantages of the algorithm in relation to the minLength parameter.

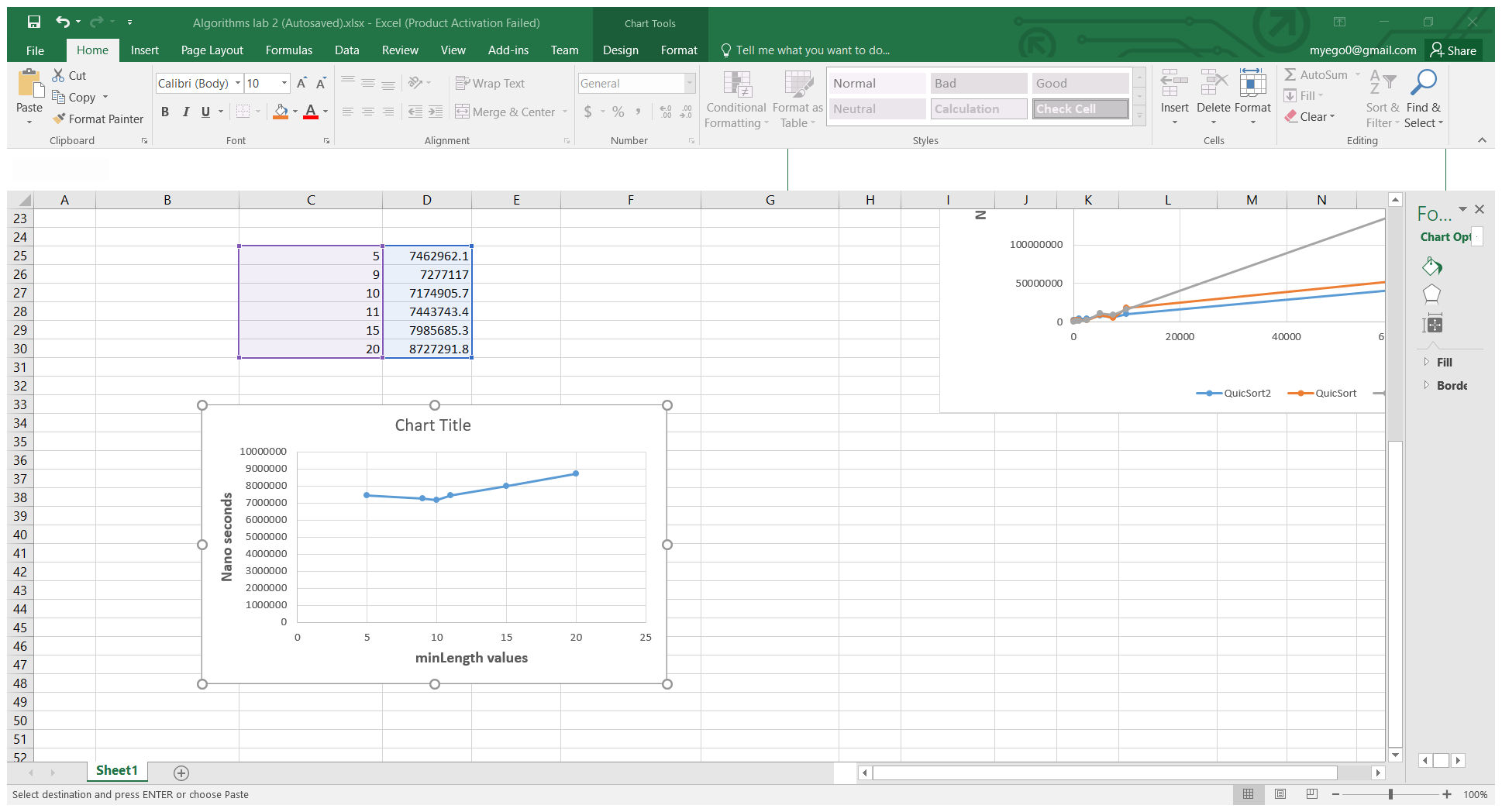
The parameter minLength determines when the InsertionSort sorting method is supposed to change QuickSort. Since the Insertion sort is quick on small arrays and has a running time of O(kn), where k equals chosen threshold value, it improves the efficiency of QuickSort algorithm when the right threshold value is chosen. According to various internet sources the threshold value is supposed to be around 10.

However, the empirical testing of QuickSort2 algorithm has shown that it’s not the value of minLength that determines the efficiency of the algorithm, but the size of array it is used on. When it is used on a short array, for example 100 elements, different minLength values have almost no influence of the algorithm performance, while on a larger array, for example 10 000 elements, a most efficient threshold value can be easily found by empirical testing.

**Task 2.**

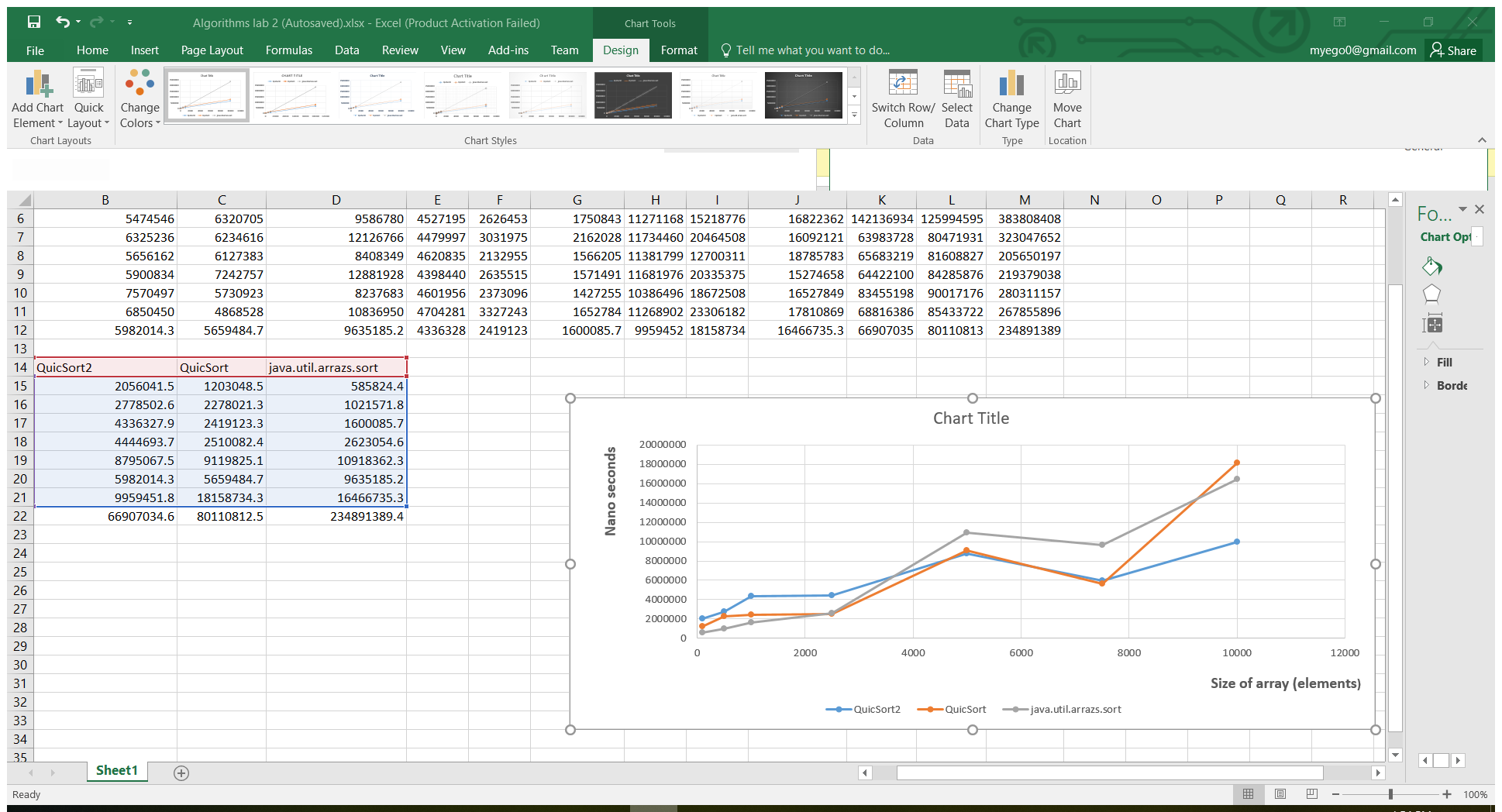
**a.**

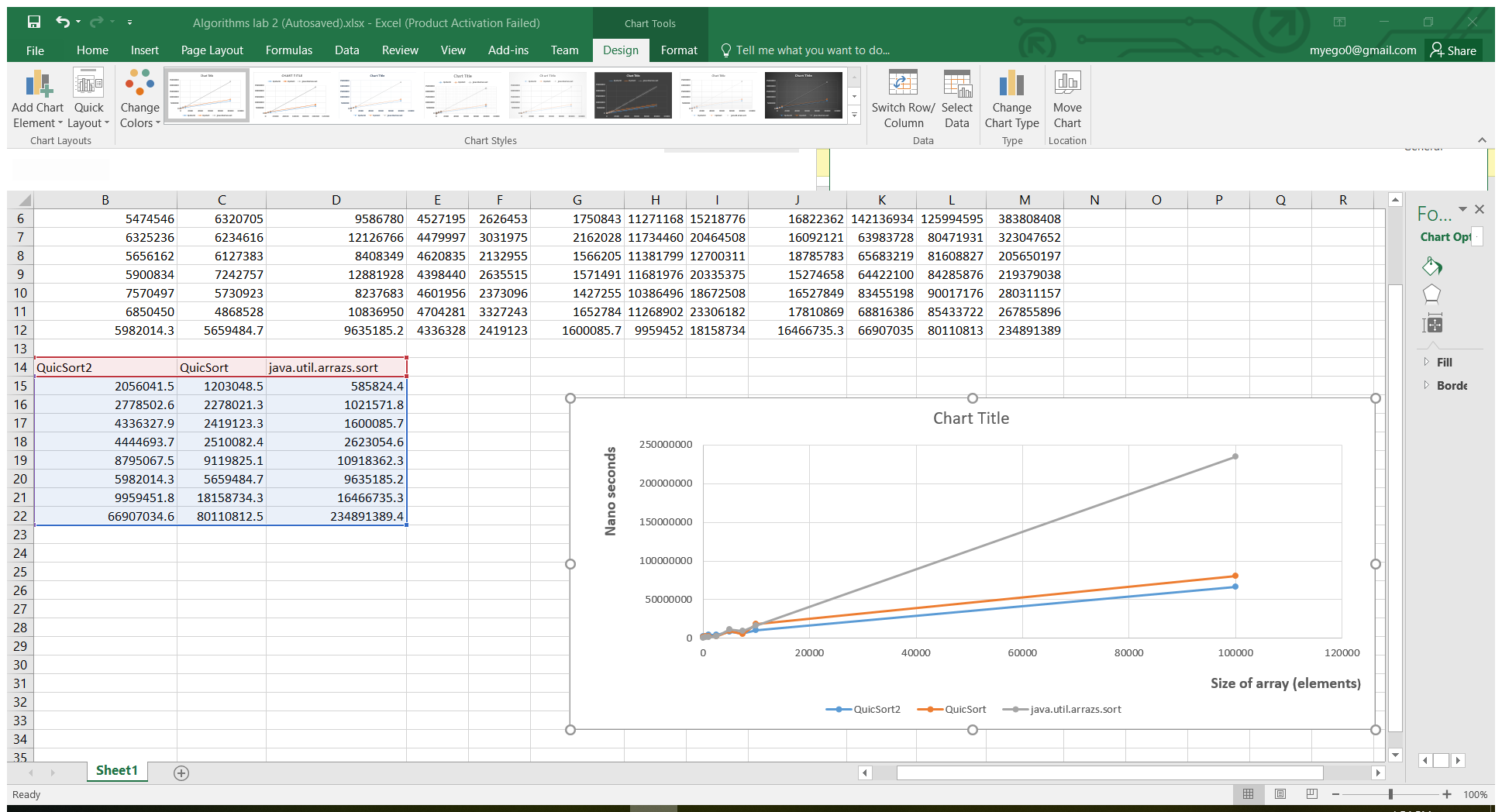
Graph below shows the average time (an average of 10 measurements) it takes to sort an array of 10 000 elements using different minLength values in QuickSort2 algotithm. It is seen that the algorithm is the most efficient when the value is 10. This is because Insertion sort is quick on small arrays and slow on larger ones.



**b.**

Graphs below show comparison of QuickSort2, QuickSort and java.util.Arrays.sort algorithms in regard to the size of of an array. For this test, the parameter minLenght in QuickSort2 was defined as 10. Average time behavious was calculated from 10 measurements for each array size.





As is seen from the graphs, on smaller arrays (up to around 1000 elements) the default algorithm is the fastest and QuickSort2 is the slowest. However, the QuickSort2 becomes the most efficient out of three at around 8000 elements and is noticeably faster as the array size increases from there.

**c.** The default sorting algorithm in java is called “Dual Pivot Quick Sort”. It is a combination of Quicksort with 2 pivots and an Insertion sort with a 47 element threshold. It was proposed by Vladimir Yaroslavskiy in 2009.

Sources:

https://en.wikipedia.org/wiki/Quicksort

<https://dev.to/s_awdesh/double-pivot-quick-sort--javas-default-sorting-algorithm-1m4>

**Task 3.**

**a.**

The code creates an ArrayList to save data of double type, and fills the array. Then parallelStream() and sorted() methods are called to sort the data, the parallel stream sorts the array in multiple threads at the same time. Then toArray(Double[]::**new**), which is a Lambda expression, is used to create a new array in which the sorted array is put, so it can be assigned to the variable Double[ ] solution.

Sources:

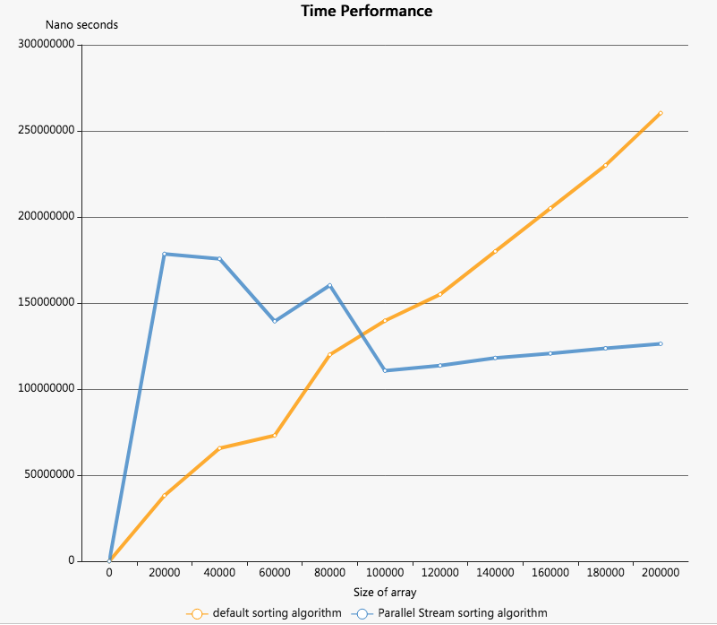
<https://www.baeldung.com/java-8-double-colon-operator>

<https://java-8-tips.readthedocs.io/en/stable/parallelization.html>

<https://java-8-tips.readthedocs.io/en/stable/streamsapi.html#to-array>

**b.**

Graphs below show comparison of default algorithm, parallel stream sorting algorithm in regard to the size of of an array. For this test, average time behavious was calculated from 10 measurements for each array size.



As is seen from the graphs, on smaller arrays (up to around 100000 elements) the default algorithm is faster than the parallel stream sorting algorithm. However, the default algorithm becomes the slower one when the number of elements over 100000.