Samuel Jackson – 2520998J

ADS2 – Assessed Exercise 1

How to run code:  
Four Java files provided, Algorithm.java, TimeSortingAlgorithm.java, TestSortingAlgorithm.java and GenerateInput.java.

GenerateInput.java can be ignored for testing algorithms, it is used for question 3, instructions there.

As long as the test files, provided by AE1 (int10.txt, int50.txt, int100.txt, int1000.txt, int20k.txt, int500k.txt, intBig.txt, dutch.txt), are all in the same file directory as the java project then this will run.

Instructions:

* Open folder as java project in Eclipse/IntelliJ (IDE).
* Open TestSortingAlgorithm.java.
* Make any adjustments to main method as desired
* Run program.

Primary method for testing all algorithms is testRunSorting(int numberOfRuns, int subarraySize), which is already in the main method, changes can be made as desired.

The main method contains the instructions and has extra commented-out code for other variations. It is currently set up to test all main sorting algorithms, bar insertion sort.

The compareAlgorithms method (in TimeSortingAlgorithm) is comprised of other private static methods, which are primarily the other sorting algorithms (which can be commented out and will not cause errors. Note that 1 is the output for an untimed algorithm).

Each algorithm contains a check to ensure that if an array is not correctly sorted then an InterruptedException will be thrown (with a stack trace which shows which array did not sort).

Findings:

It is hard to measure the lower-cases, without going into nanoseconds, so I will skip the test cases, going straight to 20k.txt and values greater than this.

The opportunity to evaluate each algorithm remains present in my opinion.

To reduce time spent talking about how slow insertion sort is, I’m going to summarise it here:

Insertion sort is by far the slowest algorithm on larger arrays. It has to deal with a significant number of a comparisons that grows hugely with each iteration within the algorithm and even more so with larger arrays. This leads the algorithm to having to deal with an unreasonable number of comparisons, with time complexity of O(n^2) on average and worst case.

Excluding insertion sort, the remaining algorithms are interesting at higher cases:

At 20,000 elements, all algorithms are very fast. However it is noticeable that insertion sort is significantly longer than the other algorithms. Merge sort is marginally slower than the quicksort algorithms.

There is a similar theme happening at 100,000 elements, as well as 500,000 and 1,000,000.

In these cases, we are seeing that merge-sort is slightly slower than the quicksort algorithms, and I believe this is a result of merge-sort having to deal with a larger space complexity O(n) on best and worst case (versus quicksort O(log n) on average). This slows down merge as it deals with more elements and comparisons when merging larger arrays.

Amongst the quicksort algorithms, insertion quicksort is the fastest because insertion sort is much faster on smaller arrays (on average between 5 and 20), this is delegating the smaller arrays to a faster algorithm and reducing the recursive calls (which are a long process as arrays get smaller). Even an array of length 5 would require 6 divisions of the array, requiring the pivot each time.

The fact that the pivot is required each time is a good reason as to why median of three is slower than insertion quicksort. Median of three has to pick three elements, sort them and pick the middle one. This does take a portion of time.

However, this is more effective than the standard quicksort because it avoids worst-case scenarios for quicksort where the right-most element (standard pivot for basic quicksort) is the largest element and there are no elements split on the pivot, effectively removing only one element from the array.

3-Way partitioning on the quicksort algorithm is actually the slowest quicksort (bar basic) and, from my perspective, this is a result of the arrays not having a large number of duplicates. Without **any** duplicates, the 3-way partitioning method becomes the same as standard quicksort, so 3-way partitioning is better with a number of duplicates. Although 3-way portioning is faster than standard quicksort in these cases, since there is a small number of duplicates.

Final test case (dutch.txt):  
This is an important test case, and more real-world relevant in my opinion because it is dealing with a number of duplicates too.

It gives the opportunity for 3-way quicksort to shine, as it is **significantly** faster than the other algorithms. This is a result of reducing the elements that have to be sorted.

3-Way partitioning is smart in the sense that, if a pivot has duplicates, it groups the pivot duplicates together and sorts partitions the elements out-with the partition of pivot duplicates, which will reduce later recursive calls, as more elements are pulled out of each recursive call than normal.

Merge sort also performs well with a large number of duplicates (performing faster than the remaining quicksort algorithms). I believe this is a result of the fact that duplicates do not affect merge-sort. We can see this pretty clearly since the array of size 500,000 (same as dutch.txt) with less duplicates has a very similar time.

The remaining quicksort methods are slower on this test case because the quicksort will return a partitioned array, and if the duplicate element ends up next to the last partition, it will create the exact same partition, not changing the order of any elements and just removing itself, which is not at all effective.

Median of three could end up picking three elements, where two or three are the same, and having a very bad partition. Standard quicksort can do the same, if the right-most element is a duplicate and the largest, being very, very slow. Insertion quicksort is marginally better off as it will not have to deal with duplicates at the lower subarrays and insertion sort is pretty efficient on duplicates.

Conclusion:

Overall, it is evident that certain algorithms thrive in different situations. The algorithms can be summarised as such:

* Small array: Insertion sort
* Consistency on large arrays: Insertion-Quicksort
* Large array with duplicates: 3-Way Quicksort

Merge Sort is a stable algorithm which is important but not for an array. This means that merge-sort is more effective than 3-way quicksort in situations when sorting things like Linked Lists or maybe an array of objects where the order is important and comparisons have been overridden within the class.

Question 3 (Pathological Input Algorithm):

GenerateInput.java is the class containing the algorithm for creating an array (and further writing to a txt file). This can be made through calling the generateOutput(int numberOfElements, String filename) method.

This is already prepared in the main method of GenerateInput (as it mostly independent of the other classes).

The algorithm functions such that it creates an array of fixed size, and it is pairing elements. It will put the largest number possible at the left side, and the next largest (one less) on the right side. This goes back and forward, decrementing each time it goes from one side to the other.

Aiming to create a worst-case scenario for median-of-three, we want to ensure that the partition is as lopsided as possible. With the median of three going to pick three integers where:

* N – largest number remaining in array (farthest left element in the array).
* K – middle number, guaranteed to be lower for integers > 2.
* L – N-1, number on right side from pair when generated.

We will have a median of three array with the choice of {N, K, L}, which sorts to {K, L, N}, picking L as the pivot point. This ensures that the only element on the right partition of the array is N. Making the array lopsided.

The result of having the array lopsided is maximising the recursive calls, removing only two elements for each call. This is because it removes pivot point as well as the only element greater (L (pivot) and N removed).

I would have tested at a greater scale about this algorithm, however I cannot avoid getting StackOverflow errors at test cases greater than 30k. If your computer is stronger, you can test in my main method of GenerateInput, there are instructions.

Note: The order in which you run functions will have an impact on the timing so theory is more important in this scenario in my opinion. (mCoding has a good video on it: <https://www.youtube.com/watch?v=ybh0GttfM8o&t=40s>).