**Metric:**

The degree of how sorted a list is can be quite abstract and thus can be interpreted in several ways. For this practical I decided to create and use a metric that is geared toward this practical. This metric takes inspiration from the Levenshtein Distance. From the viewpoint of the quick sort we know that compared sequences will always be of the same length and contain the same characters. Thus, in this situation the most efficient way to transform one sequence into another is via swapping elements (using the minimum required swaps). My metric for how sorted a list is, is based on this:

SV =

Where N is the minimum number of swaps required to sort a list and L is the length of the list. This means the Sorted Value (SV) is in the range . Thus, a sorted list is SV = 0, a slightly sorted list is 0 < SV < 0.5 and an unsorted list is 0.5 ≤ SV < 1.

**Design and Implementation:**

To get the required data I decided to design implementation that could:

* Quick Sort
* Measure the time of a Quick Sort
* Take a sorted sequence and unsort it (to a certain degree of unsorted)
* Measure the sequence using my metric
* Create a file containing the time to quick sort a sequence and how ‘sorted’ the list was

I decided to achieve this through three classes, the design of each is briefly explained below.

Quick Sort:

This class simply contains the methods to perform the sort. For my (not in-place) quick sort I decided to use the last element in the sequence as the pivot value. I decided to create an integer sorting quick sort as the focus is on performance. The more useful alternative would’ve taken comparators as a parameter and generic object lists, this would’ve had no u effects on the data.

Pair:

This class creates objects that store paired integer values. I created this class so that I could put elements and their position together into one object, making manipulation of two parallel sets of data in a synchronised fashion much easier.

Sequence Creator:

This class contains methods that allow me to get the data for the graph. The min swaps method treats the sequence like a graph of nodes and uses the relationship between nodes and their cycles to calculate the absolute minimum number of swaps to sort a list. This method is needed so we know what SV a sequence has. To get the data the methods in this class take a (sorted) sequence and records the time taken to sort it (repeated and averaged). This is repeated with the SV score of each sequence increasing with each repetition using the randomise sequence method. The SV and the time are then stored in a CSV file. The sequences are of size 1000. Choosing a ‘suitably large sequence’ is hard to justify however I chose this size as it provided enough difference in sort times to allow analysis. I decided to use 1000 different sequences for each data point to create a suitably reliable graph of how the performance changes as the SV changes.

*(Graph made using excel)*

**Results:**

Reliability – These results are reliable as I took 1000 different sequences for each Sorted Value and averaged them out. Although as the Sorted Value increases averaged sequences can be 1 or 2 out this has a limited affect as there are 1000 of them. The results in the graph were computed on my personal computer so times may vary with lab machines (or other) however the results don’t differ much between successive runs on the same machine. What also adds to these results is the fact that the machine ran this code uninterrupted with no background applications open to try and reduce the random background processing that could tarnish the results. The visible trend in the graph further suggest the results are reliable (which is different from accurate).

Graph - The graph shows a steady rapid decline and then levels at approximately 0.3 milliseconds. The graphs trend is concurrent with what we’d expect based on the metric system I have used, and the data given to it. Quick sort has a pathological case, in that when a sorted list is passed into a non-in-place quick sort using structural pivots such as the start or last index. This pathological case arises as each partitioning removes one element into the other partition. Thus, we get n elements being compared n times leading to worse case O(n2). As we move from the pathological case the performance quickly improves, this is because the more unsorted the list is, the more probable it is we will get better partitions. For the ideal performance a pivot near the median value of the list will give the best performance. Thus, it makes sense we see a decline. The rest of the graph however is very flat (The small roughness in the smooth line come from the randomness). It appears that once a certain level of unsorted-ness is reached the performance is unchanging. This is probably since they all have a similar number of recursions as the pivots are more likely to be good as the level of unsorted-ness increases.

**Conclusion on Results:**

From the graph and my metric, I can conclude that the degree to which a list is sorted generally has no effect on the performance of the quick sort. However, as the list approaches the pathological case then the performance significantly decreases.

**Evaluation:**

Overall, I think the code that is implemented for this practical was very applicable in that the data it produced allowed me to directly create a graph to analyse. In terms of the metric I believe criticism could be given to the simplicity of it however I also believe that it provides a very understandable and pragmatic way to look at the degree of sorted-ness. In terms of the data set I produce I believe that the results are very reliable, I use repetition copiously to iron out as many inconsistencies and anomalies as possible. I also believe the graph produced reflects the performance of a quicksort as one would expect.

**Usage:**

Java SequenceCreator

**Extended Work:**

To extend my work I decided to create an bubble sort and run it against the quick sort t=with the same data set to see how they differ.

Design changes – To achieve the relevant data I added an insert sort I changed the getAverage method that to get the average time of an bubble sort. The extended code now will generate data for bubble sort and I will compare that against the data from quicksort

**Results:**

Reliability - These trials were done with 5 SV increments thus there are less data points and they also have less repetitions for each SV as I did not have the time to compute for 100, this means each data point is slightly less reliable/

Graph - From the graph we can see that the bubble sorts performance decreases as the sorted value increases. This makes perfect sense as the bubble sorts number of swaps to be made is far more than the minimum (SV) and thus as the SV increases its performance should decrease. We can also see that for the quick sort pathological case the bubble sort performs better than the quick sort. This is because the bubble sort simply traverses the list n times each time with list size left unsorted decreasing. It makes no swaps during the run. There are far more outliers in this set of data as I did less repetitions leading to less reliable data points.

Conclusion:

From the graph we see that in general the bubble sort performance is far worse than that of the quick sort. And thus, the quick sort generally is far better than the bubble sort. The pathological case however does provide an interesting point that when a list is nearly sorted or completely sorted then the bubble sort performs better. This alludes to the idea that it is useful to know a little about the data before it is sorted. For suitably small lists a bubble sort would also be better however this is not shown here from my data as I have done no tests that could reflect that.