# Algorithm implementation:

Our group has decided to implement both algorithms, mergesort and quicksort in Java. The codes are appended at the end of this report.

# Generating input data:

The data are generated with a simple for loop to run from 0 to n where n = 2000, 4000, 6000, 8000, 10000. A simple randomizer is used to swap the values such that it is neither in ascending or descending order. As such, our group has generated 15 sets of input data (5 for ascending, 5 for descending and 5 for randomly order numbers). The data files are stored in a text format and are included together with the source codes for the algorithms.

# Measuring time complexity:

Both algorithms used the same 15 datasets. The number of comparisons are recorded. As CPU time may varies pending on the state of the machine it is running on, our group has repeated the test for 10 times for each algorithm, each time we recorded the CPU time and we took the average of the 10 tries. Table 1 shows the results of our tests.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Number of Comparision | | Average Execution Time | |
|  | **Merge Sort** | **Quick Sort** | **Merge Sort** | **Quick Sort** |
| Random 2000 | 19425 | **17547** | 1745148 | 2129227 |
| Random 4000 | 42825 | **38111** | 2038995 | 1800378 |
| Random 6000 | 67884 | **61908** | 6637361 | 1778965 |
| Random 8000 | 93629 | **76319** | 12530504 | 6410674 |
| Random 10000 | 120374 | **107837** | 15879367 | 14281894 |
| Ascending 2000 | **11088** | 17964 |  |  |
| Ascending 4000 | **24176** | 39917 |  |  |
| Ascending 6000 | **39152** | 63822 |  |  |
| Ascending 8000 | **52352** | 87822 |  |  |
| Ascending 10000 | **69008** | 113631 |  |  |
| Descending 2000 | **10864** | 15974 |  |  |
| Descending 4000 | **23728** | 35928 |  |  |
| Descending 6000 | **36656** | 57834 |  |  |
| Descending 8000 | **51456** | 79834 |  |  |
| Descending 10000 | **64608** | 103644 |  |  |

Analysis of results: