*Marcus Tipler*

*Cardiff University | ComSC – Abacws Building*

*[algorithms & efficienCy]*

Marcus Francis TIPLER, Computer Science at Cardiff University

**GOALS**

Build an event driven object-oriented Java program that takes as input an unsorted list of randomly generated integers, implements basic structures and uses algorithms to benchmark different sorting algorithms on the arrays. We will then proceed to compare their respective performances and provide evidence as to which of our tested sorting algorithm is best.

**MY APPROACH**

Firstly, I decided to compartmentalise individual sorting algorithms in to their own classes for future ease of use.

For the ‘Quick Sort’ algorithm I decided to use a stack-based approach, due to tests instantiating that this method drastically improves speeds and uses less memory.

For the ‘Hybrid’ sorting classes, I used subarray lists of the size of a given threshold. This means that the sub arrays are only the size of the threshold and can be sorted by themselves before parsing them back in to their primary arrays to be sorted again.

I used a combination of ‘Quick Sort’ mixed with either ‘Selection Sort’ or ‘Insertion Sort’ where ‘Quick Sort’ sorts the already sorted subarrays.

All of this was planned, and mapped using Kanban and Infinite Canvas to keep track of progress and write out algorithms in Pseudocode before programming them in (find below).

**EXTRACTS**

|  |  |
| --- | --- |
|  |  |
| ‘Quicksort Algorithm’ | ‘Quicksort Algorithm Partition’ |

|  |  |
| --- | --- |
|  |  |
| ‘Insertion-Sort Algorithm’ | ‘Selection-Sort Algorithm’ |

|  |
| --- |
|  |
| ‘Hybrid-Sorting Algorithm’ |

**BENCHMARKING**

Firstly, to benchmark the hybrid sorting algorithms whilst at their peak efficiency, we needed to test which ‘threshold’ was best for each Hybrid Sort, so ‘benchmarkComputerThreshold.java’ was made, this file outputs a ‘Comma Separated Values (CSV)’ that is readable by Microsoft Excel. Once complete, benchmark performances can be graphed accordingly:

|  |  |
| --- | --- |
|  |  |
| ‘Quick-Insertion’ threshold tests  x: threshold value.  y: time in ms. | ‘Quick-Selection’ threshold tests  x: threshold value.  y: time in ms. |

|  |
| --- |
|  |
| ‘Hybrid Sorts’ compared  x: threshold value.  y: time in ms. |

This shows that the ‘Quick Insertion’ sorting method has a better average performance at a threshold of 14 and that the most efficient ‘Quick Selection’ sorting can be is at a threshold of 8.

Now it’s possible to benchmark the algorithms themselves as follows:

|  |  |
| --- | --- |
|  |  |
| All algorithms: compared (500k random values). | Algorithms of interest: compared (1M random values). |

*(all random values are using same seed for more accurate testing)*

This shows that the performance of quicksort is outweighed by that of both Hybrid Sorts by small margins for an array size of 500k values, but the opposite effect occurs once the array size is increased to 1M random values.

**EVIDENCE**

*(loading a text file with over 500k lines was impossible for my machines, please find attached ZIP archive with file called ‘sortedProof.txt’ for full output of the Quicksort algorithm).*

|  |
| --- |
|  |
|  |
|  |
|  |

This proves the algorithm is indeed sorting all of the values as requested.

**CONCLUSION**

Overall, from what it seems thanks to the data the program provided, the Quick Sort function is much more efficient for larger datasets, although slower for the given data we were originally required to test.

|  |
| --- |
|  |