Let $n$ denote the size of any given array. For unsorted arrays, we observed that when $n$ was small, selection sort and insertion sort were faster than merge sort and quick sort. This is expected due to the recursion overhead produced by merge and quick sort. This overhead is less of an issue for larger arrays, as selection and insertion sort are both $O(n^2)$, whereas merge sort, and quick sort (given the fact that the median element is clearly a ‘good’ pivot as the partition lengths must always be $\le\dfrac{3n}4$) are both $O(n\log{n})$, and thus as expected, merge and quick sort were much faster for larger arrays. Interestingly, merge sort was faster than quick sort for larger arrays, but slower for smaller arrays. This is likely because merge sort usually makes more recursive calls and thus produces more overhead. Also, quick sort operates under tail recursion, thereby optimising this issue. For larger arrays, clearly both algorithms will have many recursive calls, but on each call, quick sort has more operations due to the number of while loops to iterate through. Generally, we observed selection sort to be slower than insertion sort, although this trend was not as clear-cut for smaller arrays. This trend is likely because selection sort continually scans the entire array, whereas insertion sort only needs to inspect the unsorted region, hence requiring fewer operations. This also explains why in sorted arrays, selection sort was significantly slower than the other algorithms, and why insertion sort was the fastest algorithm in this scenario. Regarding reversed arrays, similar, but less consistent trends were observed compared to randomly unsorted arrays. This is expected as this is simply the worst-case scenario of all sorting algorithms, whereas the sorted case is typically the best-case scenario.