Karlis Siders (2467273S)

Part 0: How To

All work is contained in 2 files: AE1.java and TimeSortingAlgorithms.java, both of which are runnable. AE1.java contains all algorithms and some testing code behind comments, and TimeSortingAlgorithms.java contains methods for reading arrays from text files in a 'num' folder, which should be in the same directory as the 'src' folder, and methods for timing all algorithms (Insertion Sort is commented to save minutes of your time).

Easiest way to run the code:

1) Make a project called 'ADS-AE1' (I don't know whether the name is necessary) in Eclipse
2) Add 'num' folder and replace 'src' folder from zip file
3) Edit main methods in AE1.java and TimeSortingAlgorithms.java as necessary
4) Run TimeSortingAlgorithms.java for timing, AE1 for testing specific methods/algorithms.

Part 1: The Algorithms

A: The standard Quicksort algorithm choosing the last element as a pivot while partitioning.

B: An optimisation of Quicksort which stops sorting subarrays with fewer than *k* elements and then calls Insertion Sort on the whole array. Implemented by a helper method, which calls the optimised Quicksort, which, in turn, calls itself recursively, finally allowing the helper method to call Insertion Sort on the whole array.

C: An optimisation of Quicksort which chooses the pivot as the median of three values in the array (out of the leftmost, rightmost, and middle). Places the pivot at the end of the array, as is customary.

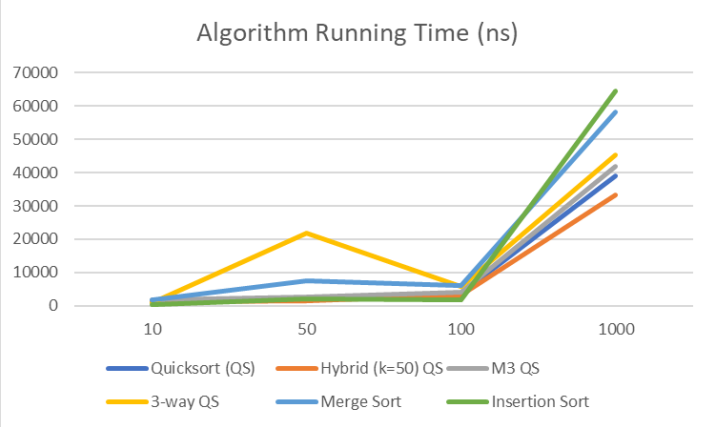
D:

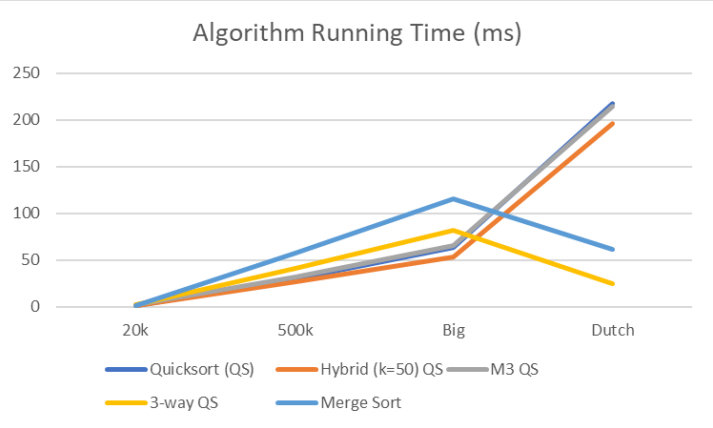
An optimisation of Quicksort which partitions the whole array into 3 parts: less than the pivot, equal to it, and greater than it.
On a broad scale, the algorithm checks whether the left index is less than the right index, and, if so, calls the **partition** method, which returns two indices: the index where the middle part starts and the index where it ends. The two indices are then used to call Quicksort recursively on the **first** and **third** parts (since the middle part, i.e., occurrences of the same number, is already sorted).
To be more precise, the **partition** chooses the last element as the pivot **x** (which could be further optimised, e.g., with the median-of-three method) and keeps track of two indices **less** and **greater** that start on opposite sides of the array (left and right, respectively) and a third index **i** to help update the previous two. **i** then goes through the whole array, comparing the current value to the pivot: if the value is less, it swaps the values at indices **i** and **less** and advances both indices; if the value is more, it swaps the values at indices **i** and **greater** and decreases the **greater** counter; otherwise, it advances **i** and looks at the next element in the array/subarray. Finally, it returns the **less** and **greater** counters as an array of size two.

Part 2: The Comparison

Algorithm running times taken as an average of 10 runs.

	Array size (time measurement)						Array name (time)	
Algorithm	10 (ns)	50 (ns)	100 (ns)	1000 (ns)	20k (ms)	500k (ms)	Big (ms)	Dutch (ms)
Quicksort (QS)	1200	1900	3800	38900	1	30	63	218
Hybrid (k=50) QS	1200	1400	3300	33300	1	27	53	196
M(edian of) 3 QS	1900	2800	4100	42000	2	32	65	215
3-way QS	1100	21800	5900	45400	2	41	82	25
Merge Sort	1700	7600	6200	58200	1	57	116	61
Insertion Sort	500	2100	1800	64400	24	16551	69676	101657





The most notable difference is between the simple, non-recursive Insertion Sort and all of the other sorting algorithms: when there are fewer than 50 elements, Insertion Sort sorts the array the fastest; however, when there are more than 1000 elements in the array, it is significantly slower, taking minutes to complete. This is because Insertion Sort has the least overhead out of all of the above algorithms, which makes a big difference when the array to sort is small or nearly sorted, while it is also not as well optimised as the other algorithms for large arrays.

Merge Sort is almost always the slowest or second slowest algorithm compared to its competitors; however, it is significantly faster at sorting the Dutch array than every other algorithm except the one specifically designed to sort it (3-way QS).

Standard Quicksort's running time increases as the number of elements in the array increases. More efficient than the aforementioned algorithms (except for Dutch array), but it can be optimised further.

The hybrid implementation of Quicksort, which calls Insertion Sort instead of sorting subarrays with fewer than *k* elements, performs extremely well in almost all cases as it plays to each sorting algorithm's strengths: Insertion Sort with only a few elements to sort and Quicksort doing the grunt of the work.

The Median of Three (M3) Quicksort does surprisingly badly in most cases, but it is marginally better at sorting the Dutch array. This could be explained by unfortunate array inputs, as the median could

correspond to the same pivot chosen by standard Quicksort, which would then mean that the only difference is that M3 has more needless comparisons than its competitors.

Finally, the Three-Way Quicksort is marginally better than other Quicksort implementations at sorting smaller inputs and extremely faster (almost 90% faster than standard Quicksort) at sorting the Dutch array, which contains many duplicates and is the reason why this optimisation works so well as the Three-Way Quicksort gathers all elements equal to the pivot in the middle of the (sub)array.

Part 3: The Pathological Input

A pathological input for a median-of-three Quicksort would be one that always chooses the second largest or second smallest element in a (sub-)array (but not the largest or smallest since they cannot be the median of three without repetition) in order for the partitioning to be as inefficient as possible (partitioning only one or two elements on one side and all of the rest on the other). Thus, the first step is to put two extreme values (I chose the highest) in two of the spots being analysed by the partition method. The Pathological Input generational algorithm makes a new, already sorted array, which then it copies into the pathological one by iterating over the first half of it. Every other iteration, it sets the value of the previous index in sorted order (by just copying the sorted array) and the value of the current index as the odd values of the second half of the sorted array. Additionally, on every iteration, it sets the second half of the array as a sorted array of even numbers. This way, it is highly likely that my implementation of the median-of-three Quicksort reaches *stack overflow* with larger input sizes.