Monica Martinez
Data Structures
Proj. Klukowska

Project II: Benchmarking Sort Algorithms
Sort Algorithms: Selection Sort, MergeSort, and QuickSort.

I.

Selection Sort works by dividing the array into two distinct sections, sorted and unsorted. Initially, the sorted array is empty however the algorithm iterates through the unsorted array and adds the smallest value it finds into the sorted array until the unsorted array is empty. The Selection Sort algorithm used in the Benchmark was implemented using an iterative approach and based on the number of tests conducted with varying number of elements (from 10000 to 100000,) the sort exhibit quadratic behavior as expected. Given the nature of an algorithm of O(N^2), no further tests were conducted with number of elements ranging between 150000 to 1000000.

Quick Sort works by implementing a divide and conquer approach that partitions the array into two sections based on whether the values are less than or greater than a pivot. That is to say, after a pivot is selected values are constantly swapped in order to ensure that everything on the left of the pivot is less than the pivot and everything on the right of the pivot is greater than the pivot. The Quick Sort algorithm implementation was based on the implementation found in the OpenDSA website. This implementation used a recursive approach and it also uses the middle element of the array as its pivot. The expected runtime of this algorithm was O(Nlog2 N) which was confirmed by the various tests conducted using an array size of range between 10000 and 1000000.

Modifications were attempted in order to determine whether pivot selection played an important role in the overall time efficiency of the algorithm. The control implementation of quicksort uses a *findPivot()* method that returns the middle index as the pivot. Three experimental implementations used different methods of finding pivot positions: First element as pivot, Last element as pivot, and the Median of Three elements as pivot. The median of three implementation uses the median value of the first, last, and middle elements as pivot. Based on the results (charts 3 and 4*) it is difficult to determine exactly which implementation is superior as all four implementations seem to perform within a small difference from one another. For example, in the case of 100000 array elements, all four implementations had runtime between 14 and 19 milliseconds. On the first run it was Left as Pivot that had the slowest runtime (19ms) however on the second test, Left as Pivot actually had the fastest runtime (14ms.) Therefore the effects of pivot selection seem inconclusive. A suggestion would be to increase the number of tests to include partially sorted arrays, sorted arrays, and also implement a Median of Three method that uses random array elements instead of first, middle, and last.

Merge Sort also works by implementing a divide and conquer approach however, merge sort literally divides the array into subarrays and then once those subarrays are sorted, merges them back into the original array. The Merge Sort algorithm implemented was based on the implementation found in the OpenDSA website. It utilizes a recursive approach that also uses a temporary array in order to place a copy of the elements during the merging process. The expected runtime of this algorithm was

O(Nlog2 N) and was also confirmed in the various tests conducted using number of elements ranging between 10000 and 1000000.

Modifications were attempted in order to determine whether the removal of a temporary array dependence was relevant in the total efficiency of the algorithm. Given the fact that the usage of a temporary array to store elements during merging would require less memory space, it was assumed that its removal would improve efficiency. As the tests demonstrate (chart 5 and 6*,) both Temp-array and In-Place implementations of merge sort performed similarly when given number of elements between 10000 and 100000, with the exception of an array of size 100000, in which case the Temp-array implementation performed significantly slower when compared to the In-Place implementation. However, when given array sizes of range 150000 to 1000000, the In-Place implementation demonstrated quadratic behavior whereas the Temp-Array implementation performed as expected O(Nlog2 N.) These tests help delineate a clear difference between Memory and Time when determining efficiency. If one were more interested in a conserving space, then the In-Place implementation is ideal but if one is interested in conserving time, then the Temp-Array implementation should be preferred.

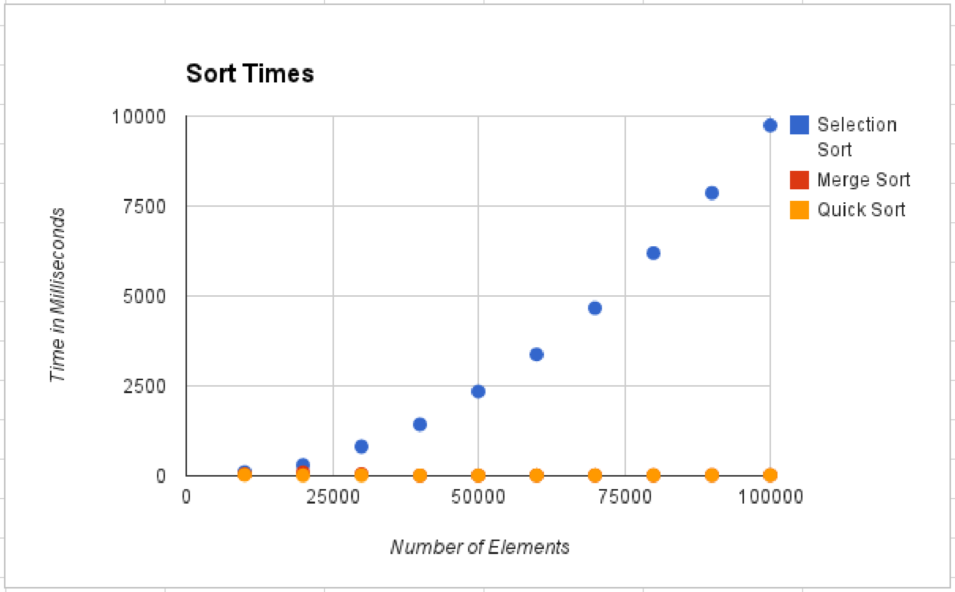
II.

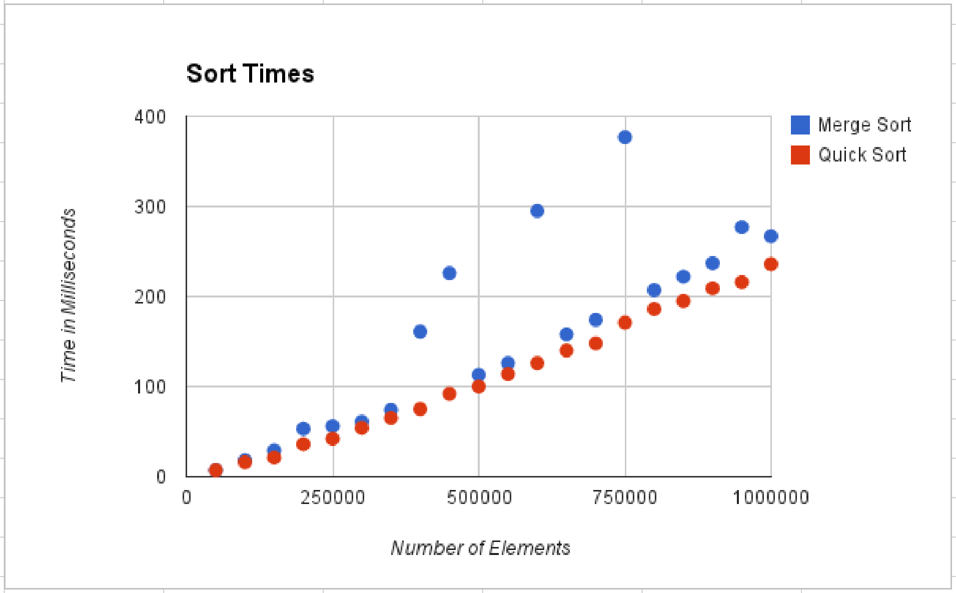
Graphs and Charts (see attached spreadsheets)

*Note: The chart numbers are determined by their sequential order of appearance in the attached spreadsheets.

Sorting Benchmark				
Num. of Elements	Selection Sort	Merge Sort	Quick Sort	
10000	104	42	25	A
20000	294	95	9	
30000	811	46	11	
40000	1426	6	5	
50000	2344	7	6	
60000	3369	10	8	
70000	4666	11	9	
80000	6190	13	11	
90000	7865	17	13	
100000	9746	17	14	

Num. of Elements	Merge Sort	Quick Sort
50000	7	7
100000	18	16
150000	29	21
200000	53	36
250000	56	42
300000	61	54
350000	74	65
400000	**161**	75
450000	**226**	92
500000	113	100
550000	126	114
600000	**295**	126
650000	158	140
700000	174	148
750000	**377**	171
800000	207	186
850000	222	195
900000	237	209
950000	277	216
1000000	267	236

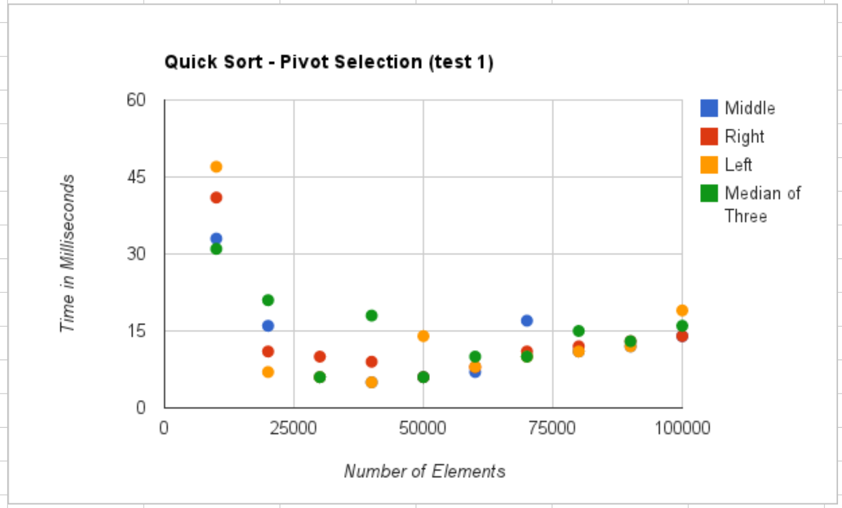




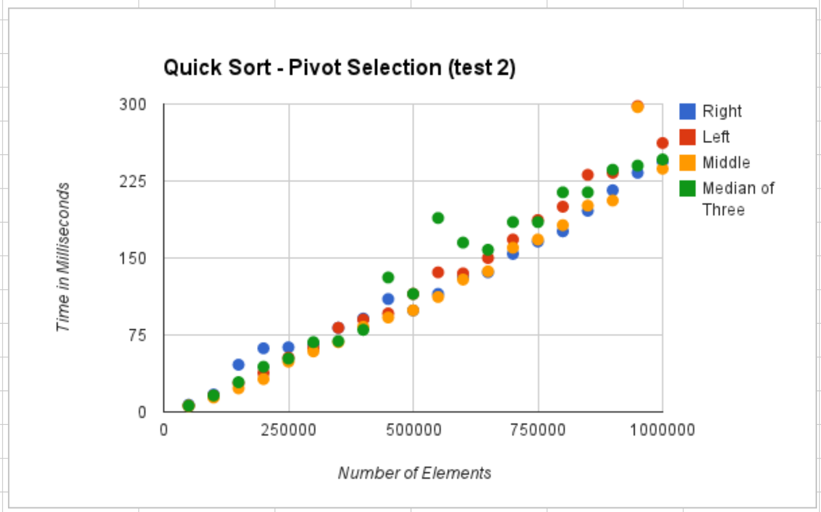
Modifications - Sorting Benchmark

QUICK SORT

Num. of Elements	Middle	Right	Left	Median of Three
10000	33	41	47	31
20000	16	11	7	21
30000	6	10	6	6
40000	5	9	5	18
50000	6	6	14	6
60000	7	8	8	10
70000	17	11	10	10
80000	11	12	11	15
90000	12	13	12	13
100000	14	14	19	16

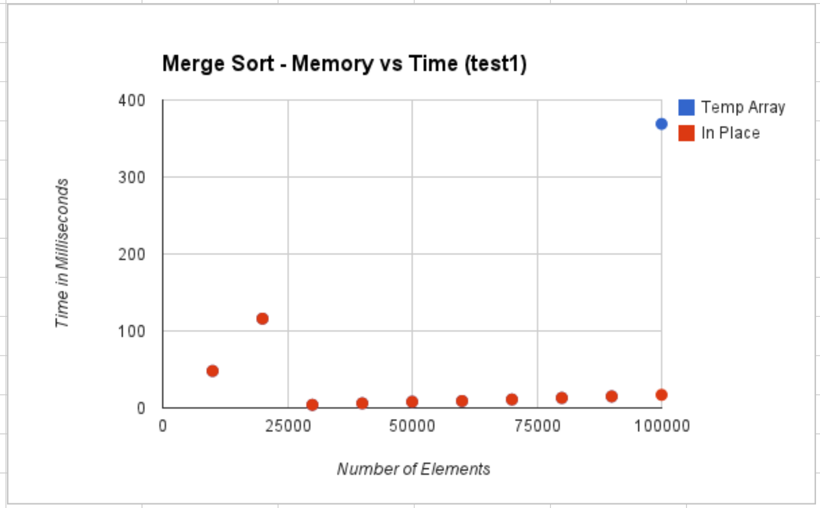


Num. of Elements	Right	Left	Middle	Median of Three
50000	7	6	6	6
100000	17	16	14	16
150000	46	28	23	29
200000	62	38	32	44
250000	63	53	49	52
300000	66	62	59	68
350000	82	82	68	69
400000	91	90	83	80
450000	110	96	92	131
500000	99	115	99	115
550000	115	136	112	189
600000	131	135	129	165
650000	136	150	137	158
700000	154	168	160	185
750000	166	187	168	185
800000	176	200	182	214
850000	196	231	201	214
900000	216	233	206	236
950000	233	298	297	240
1000000	244	262	237	246



MERGE SORT

Num. of Elements	Temp Array	In Place
10000	48	48
20000	116	116
30000	4	4
40000	6	6
50000	8	8
60000	9	9
70000	11	11
80000	13	13
90000	15	15
100000	369	17



Num. of Elements	Temp Array	In Place
50000	7	67
100000	15	367
150000	26	816
200000	54	1538
250000	56	2519
300000	60	3640
350000	83	5131
400000	157	6818
450000	211	8732
500000	109	10813
550000	124	13366
600000	294	16086
650000	172	19057
700000	193	22128
750000	272	25842
800000	225	29745
850000	219	33267
900000	238	37622
950000	274	41878
1000000	265	46677