Ryan Skinner
CS3310
10/7/18

For this project I tested the efficiency of 5 different sorting algorithms implemented in java. I tested each with both sorted and unsorted integer arrays ranging in size from 2^1 to 2^16. Each algorithm was tested 10,000 times and an average time was recorded in nanoseconds. The testing was run on my home computer, it has a Ryzen 1500 CPU clocked at 3.2 GHz, it has 6 cores (12 threads) but only one thread was used at a time, so for purposes of testing it was not useful. Below is the table with my results, the raw data can be found in 'output-orig.csv', this table has been truncated because, for reasons outlined later, the first results are far off and not reliable.

		2^6	2^7	2^8	2^9	2^10	2^11	2^12	2^13	2^14	2^15	2^16
Insertion	unsorted	1441	4417	17871	45516	170842	655019	2571969	10108335	38293654	151545390	608319266
Merge	unsorted	3007	10718	48699	154011	596787	2265188	7889893	30877226	122513912	488718408	1979353258
Quick	unsorted	771	2651	10138	34267	129916	474811	2006955	7804292	30002105	119202061	474748253
Quick2	unsorted	457	1084	2386	7233	17629	59359	160443	395437	905178	2028386	4147820
Quick3	unsorted	2110	5143	11558	25795	56443	122128	261864	559212	1188054	2562784	5413608
Insertion	sorted	131	221	412	815	1551	2936	5778	11534	22895	45489	90720
Merge	sorted	2408	9853	41018	153036	590637	2232468	7767762	30593733	121519422	485572859	1972983720
Quick	sorted	149	223	439	796	1540	2925	5827	11533	22911	45440	90870
Quick2	sorted	1723	5401	17786	61860	227770	872323	3410282	13467886	53555171	213792172	853615877
Quick3	sorted	1120	2477	5427	11617	24838	52786	112093	234848	495316	1037624	2177350

An issue with my code caused huge errors in lower sized arrays. Running the loop 2^22 times to take the average execution time instead of 10,000 times created much better numbers for arrays less than 2^6 in size. The table of those arrays are posted below. I came to the number 2^22 by running a loop, each time doubling it to see when the execution time would average out for each size array. The reason I didn't collect an average for all sized arrays with this number is it would take way too long and didn't improve the data very much, if at all.

		2^1	2^2	2^3	2^4	2^5	2^6	2^7
Insertion	unsorted	35	25	51	86	293	1087	3609
Merge	unsorted	38	77	146	377	949	2692	10257
Quick	unsorted	19	33	44	80	201	899	2786
Quick2	unsorted	23	29	45	79	260	449	1248
Quick3	unsorted	27	40	62	116	726	2173	5238
Insertion	sorted	20	20	33	41	55	92	153
Merge	sorted	30	62	123	316	823	2452	9943
Quick	sorted	19	20	33	41	56	91	154
Quick2	sorted	25	29	42	58	428	1879	6161
Quick3	sorted	28	41	80	190	486	1148	2557

Quicksort2 (using insertionsort for n<=16) and quicksort3 (using random pivot index) seemed to have a huge improvement over quicksort1, in larger sized arrays, 2^6 for quicksort2 and 2^8 for quicksort3. However this effect is only seen in unsorted arrays. It seems like, for unsorted arrays, quicksort2 had the best results. However for sorted arrays quicksort2 was the second worst option, and quicksort3 was better. In sorted arrays insertionsort had almost identical times as quicksort1, they both were the fastest. Merge sort had the worst time regardless of the sortedness of the array. If I had to pick an algorithm and I didn't know if the list was presorted or not, I would pick quicksort3, otherwise quicksort2 would be the best option.

I assume there was an error in my merge sort code, because it should not have operated so poorly. Although the sortedness of the array did not effect the execution time, which is true of the merge sort algorithm. Otherwise I think my implemented algorithms operate within a reasonable error of the time required to execute each operation.

Average theoretical	Quicksort	Mergesort	Insertionsort
My data	Quicksort	Insertionsort	Mergesort