Mike Ogrysko – Lab 4 - Analysis

Data Structures: Description and Justification

In my application, I have 7 sorts:

1. QuickSort with first item as pivot
2. QuickSort with first item as pivot and 100 as the stopping case with Insertion Sort
3. QuickSort with first item as pivot and 50 as the stopping case with Insertion Sort
4. QuickSort with median-of-3 as pivot
5. Natural MergeSort using a linked list to process the list
6. Traditional MergeSort using a linked list to process the list
7. Natural MergeSort using an array to process the list

All sorts except for #7 are recursive. I chose the recursive code options for the simple fact that they seemed to be most readily available for insertion in this assignment. I pulled an example of an iterative Natural MergeSort and found that the functions were similar in purpose – one function to create the partitions and another function to merge them back together. I assume iterative versions of the quicksorts would be constructed in a similar way. In my testing, I found that the iterative version was much more efficient for random and reverse order files and files with duplicates. I experienced the same efficiency when processing the ascending order input files. I discuss this in more detail in the efficiency section.

For the quicksorts that use stopping cases of 100 and 50 (2 and 3), I ultimately implemented an iterative version of an insertion sort. I did try a recursive insertion sort (and left the code in my program) but found that the sorts would not process the input files with higher numbers. As a result, my quicksorts that have 100 and 50 stopping cases use an iterative insertion sort. The comparison and swap numbers for both versions of the insertion sort are in the Test Cases section below.

Using my mergesorts as a basis, it seems like the iterative versions of the sorts take less memory and, in the case of the Natural MergeSort, performed fewer comparisons and swaps. For these two sorts, the only difference was found when processing the ascending ordered files. Here, the numbers were identical, which makes sense given the nature of a Natural MergeSort. When I processed the iterative and recursive versions of the insertion sort, I found that the recursive version produced fewer comparisons and swaps. However, I was unable to run the recursive version of the insertion sort (within the quicksort) on larger input files. Again, this seems to be an issue of memory.

For testing purposes, I have used 32 test files. The files are of size 50, 1000, 2000, 5000, 10000, 15000, 20000, and 40000 (8) and are random, reverse, ascending, and duplicates (4). I used the files provided as a basis and created additional files as necessary.

Using code found online, I created a QuickSort class that contains all of my quicksorts, as well as my insertion sorts (iterative and recursive). I also created a LinkedList class, which uses a Node class for my Natural and Traditional MergeSorts. In addition, I created a MergeSortArray class for my iterative array based Natural MergeSort. All code pulled from online has been cited within the code.

Test Cases: Comparisons and Swaps

		QuickSort (1)		QuickSort Stop 100 (2)		QuickSort Stop 50 (3)		QuickSort Median (4)		Natural MergeSort (5)		Traditional MergeSort (6)		Natural MergeSort w Array (7)	
	#	**Comps**	**Swaps**	**Comps**	**Swaps**	**Comps**	**Swaps**	**Comps**	**Swaps**	**Comps**	**Swaps**	**Comps**	**Swaps**	**Comps**	**Swaps**
Random	**50**	222	68	676	632	676	632	189	132	455	414	225	117	243	97
	1000	11295	2174	1499934	228372	1554558	228218	4579	3562	174645	173650	8705	4319	9210	4091
	2000	25415	4920	5843026	813172	6053306	811902	9614	7640	676818	674824	19436	9609	20456	9252
	5000	75744	13368	30001728	2545156	31087376	2545127	26198	20632	4195289	4190298	55160	27027	59312	25426
	10000	156574	29658	129325977	11300578	130866054	11299302	53797	43728	16691319	16681330	120483	59112	128866	56009
	15000	267916	46207	349456431	44643561	353299643	44643199	82613	67691	37483797	37468809	189370	93517	197769	89306
	20000	351511	63501	631553963	79425333	634584692	79424851	112323	92259	NA	NA	260994	128483	277375	122139
	40000	725119	134287	2108570851	176647589	2118589476	176647589	241572	193875	NA	NA	561851	276715	594353	264067
Reverse	**50**	1225	25	1225	1225	1225	1225	179	89	1225	1225	133	133	182	133
	1000	499500	500	1891053	497504	1944778	497504	3054	1524	499500	499500	4932	4932	5931	4932
	2000	1999000	1000	7787053	1995004	7890778	1995004	6114	3048	1999000	1999000	10864	10864	12863	10864
	5000	12497500	2500	49475053	12487504	49728778	12487504	17259	8406	12497500	12497500	29804	29804	34803	29804
	10000	49995000	5000	198955053	49975004	199458778	49975004	34519	16810	NA	NA	64608	64608	74607	64608
	15000	NA	NA	448435053	112462504	449188778	112462504	48454	23884	NA	NA	102252	102252	117251	102252
	20000	NA	NA	NA	NA	NA	NA	69039	33618	NA	NA	139216	139216	159215	139216
	40000	NA	NA	NA	NA	NA	NA	138079	67234	NA	NA	298432	298432	338431	298432
Ascending	**50**	1225	0	49	0	49	0	155	62	49	0	153	0	49	0
	1000	499500	0	1393550	0	1447275	0	2555	1022	999	0	5044	0	999	0
	2000	1999000	0	5792050	0	5895775	0	5115	2046	1999	0	11088	0	1999	0
	5000	12497500	0	36987550	0	37241275	0	14760	5904	4999	0	32004	0	4999	0
	10000	49995000	0	148980050	0	149483775	0	29520	11808	9999	0	69008	0	9999	0
	15000	112492500	0	335972550	0	336726275	0	40955	16382	14999	0	106364	0	14999	0
	20000	NA	NA	NA	NA	NA	NA	59040	23616	19999	0	148016	0	19999	0
	40000	NA	NA	NA	NA	NA	NA	118080	47232	39999	0	316032	0	39999	0
Duplicates	**50**	304	44	609	560	609	560	197	127	364	320	222	102	238	84
	1000	10479	2181	1077469	128173	1209322	128173	4647	3579	163802	162808	8670	4284	9204	4079
	2000	24051	4721	4916362	528884	5174855	528372	10183	7610	657840	655846	19339	9567	20436	9167
	5000	71718	13559	31318865	3014549	32490242	3014056	26113	20852	4152656	4147666	55324	27143	59313	25491
	10000	147865	30022	135145920	13870804	138168270	13870804	54449	43743	16507667	16497677	120413	59077	128738	55877
	15000	249781	45304	305705794	31059989	311298049	31059989	86439	68024	37204535	37189546	189257	93298	197619	89319
	20000	327447	62211	NA	NA	NA	NA	167902	135755	NA	NA	260824	128152	277526	121886
	40000	705017	125032	NA	NA	NA	NA	255217	197216	NA	NA	561644	276300	594876	263591

Test Cases: Comparisons and Swaps – Insertion Sort Recursion

These are the comparison and swap numbers for quicksorts with the stopping cases that use the recursive insertion sort. For the sake of comparison, I have included the comparisons and swaps for the quicksorts using the iterative insertion sort.

		QuickSort Stop 100 (2)		QuickSort Stop 100 Recursive IS		QuickSort Stop 50 (3)		QuickSort Stop 50 Recursive IS	
	#	**Comps**	**Swaps**	**Comps**	**Swaps**	**Comps**	**Swaps**	**Comps**	**Swaps**
Random	**50**	676	632	676	632	676	632	632	632
	1000	1499934	228372	662777	228372	1554558	228218	666451	228218
	2000	5843026	813172	2410745	813172	6053306	811902	2417128	811902
	5000	30001728	2545156	7611210	2545156	31087376	2545127	7627070	2545127
	10000	129325977	11300578	NA	NA	130866054	11299302	NA	NA
	15000	349456431	44643561	NA	NA	353299643	44643199	NA	NA
	20000	631553963	79425333	NA	NA	634584692	79424851	NA	NA
	40000	2108570851	176647589	NA	NA	2118589476	176647589	NA	NA
Reverse	**50**	1225	1225	1225	1225	1225	1225	1225	1225
	1000	1891053	497504	991953	497504	1944778	497504	995728	497504
	2000	7787053	1995004	3988953	1995004	7890778	1995004	3992728	1995004
	5000	49475053	12487504	24979953	12487504	49728778	12487504	24983728	12487504
	10000	198955053	49975004	NA	NA	199458778	49975004	NA	NA
	15000	448435053	112462504	NA	NA	449188778	112462504	NA	NA
	20000	NA	NA	NA	NA	NA	NA	NA	NA
	40000	NA	NA	NA	NA	NA	NA	NA	NA
Ascending	**50**	49	0	0	0	49	0	0	0
	1000	1393550	0	494450	0	1447275	0	498225	0
	2000	5792050	0	1993950	0	5895775	0	1997725	0
	5000	36987550	0	NA	NA	37241275	0	NA	NA
	10000	148980050	0	NA	NA	149483775	0	NA	NA
	15000	335972550	0	NA	NA	336726275	0	NA	NA
	20000	NA	NA	NA	NA	NA	NA	NA	NA
	40000	NA	NA	NA	NA	NA	NA	NA	NA
Duplicates	**50**	609	560	560	560	609	560	560	560
	1000	1077469	128173	373179	128173	1209322	128173	382155	128173
	2000	4916362	528884	1570040	528884	5174855	528372	1578656	528372
	5000	31318865	3014549	9003334	3014549	32490242	3014056	9019941	3014056
	10000	135145920	13870804	NA	NA	138168270	13870804	NA	NA
	15000	305705794	31059989	NA	NA	311298049	31059989	NA	NA
	20000	NA	NA	NA	NA	NA	NA	NA	NA
	40000	NA	NA	NA	NA	NA	NA	NA	NA

Observations on Sorts

QuickSort (1)
We see that standard QuickSort with the first item as the pivot performs the fewest number of comparisons when the input files are presented in random order. Ascending and reverse ordered files result in the largest number of comparisons. In fact, the comparisons are the same for ascending and reverse ordered files running this sort. When processing input files with duplicates, the number of comparisons is comparable to (but higher than) the number of comparisons realized when processing randomly ordered files.

The number of swaps is zero when processing input files that are provided in ascending order. The number of swaps when processing reversely ordered files is lower than the number of swaps when processing randomly ordered input files and input files containing duplicates (which are comparable).

Input files provided in reverse and ascending order and sized 20K and 40K (as well as 15K for reverse) could not be processed, more than likely because of a memory issue.

Based on only the comparisons and swaps, this sort seems best suited for a randomly ordered file.

QuickSort Stop 100 (2)
We see that QuickSort with the first item as the pivot and a stopping case of 100 performs the fewest number of comparisons when the input files are presented in ascending order. When the number of items to be processed in ascending ordered files is 50 or less, the number of comparisons is less than the number of items to be sorted (assuming no duplicates). This is due to the insertion sort, which sorts the whole list in this case. Reversely ordered files result in the largest number of comparisons. Randomly ordered files and files with duplicates result in similar numbers of comparisons with a lower number of comparisons in files with duplicates.

The number of swaps is zero when processing input files that are provided in ascending order. The number of swaps when processing reversely ordered files is lower than the number of swaps when processing randomly ordered input files and input files containing duplicates (which are comparable).

Input files provided in reverse and ascending order and with duplicates and sized 20K and 40K could not be processed, more than likely because of a memory issue.

Comparing the insertion sorts, the number of comparisons and swaps was lower on the recursive version; however, the recursive version would not run on input files with higher numbers.

Based on only the comparisons and swaps, this sort seems best suited for ascending ordered files under 100 items.

QuickSort Stop 50 (3)
For files sizes 50 and under, the number of comparisons and swaps is identical to the QuickSort with a stopping case of 100. Again, the number of comparisons is due to the insertion sort, which sorts the whole list in this case. For all other file sizes, the numbers are comparable, though slightly higher.

Based on only the comparisons and swaps, this sort seems best suited for ascending ordered files under 100 items.

QuickSort Median (4)
Of all the quicksorts, the QuickSort with a median-of-three as a pivot produces the lowest numbers of comparisons for any ordered file at any size. The numbers of comparisons are the best when the files are in ascending order; however, all numbers are comparable regardless of the order of the files.

Of all sorts compared, this sort is the only sort where the number of swaps is not zero for an ascending order file. That said, the number of swaps is lower for randomly and reversely ordered files, as well as files with duplicates, for each size. Of the files, the number of swaps is lowest for ascending ordered files.

This sort ran for each file size, regardless of order.

Based on only the comparisons and swaps, this sort seems best suited for ascending ordered files and produces consistently produces low numbers regardless of file order (especially among quicksorts).

Natural MergeSort (5)
The Natural MergeSort built recursively and with a linked list produced the lowest comparisons and swaps when the files where in ascending order. In this case, the number of comparisons is one less than the number of items in the file and the number of swaps is zero.

For all other cases, the numbers of comparisons and swaps were nearly identical within a given file order. Randomly ordered and files with duplicates produced similar numbers of comparisons and swaps with the files with duplicates producing lower numbers. Reversely ordered files produced the largest number of comparisons and swaps.

This sort did not run for 20K and 40K files in randomly and reversely ordered files, as well as files with duplicates. Also, this sort did not run on reversely ordered files of size 10K and 15K.

Based on only the comparisons and swaps, this sort seems best suited for ascending ordered files.

Traditional MergeSort (6)
While the Traditional MergeSort had higher comparisons on ascending ordered files than the other merge sorts (and all sorts except for the first quicksort), the number of swaps was still zero. The number of comparisons was lowest for reversely ordered files. Randomly ordered files and files with duplicates produced similar numbers of comparisons.

The numbers of comparisons and swaps were the same for reversely ordered files. The numbers of swaps were nearly the same regardless of the order of the file.

With the exception of input files provided in ascending order, this sort seemed to produce the lowest numbers of comparisons and swaps among the mergesorts tested.

This sort ran for all file sizes and orders.

Based on only the comparisons and swaps, this sort seems best suited for reversely ordered files as it produced the best numbers of the mergesorts.

Natural MergeSort w Array (7)
Much like the Natural MergeSort built recursively with a linked list, the numbers of comparisons are one less than the number of items in the files and the numbers of swaps are zero.

This sort produced similar comparison and swap numbers for randomly ordered files and files with duplicates. Input files provided in reverse order produced lower numbers of comparisons (by nearly half) than randomly ordered and files with duplicates and slightly higher numbers of swaps.

This iterative version seemed to produce lower numbers of comparisons and swaps for every file size in every order (except for ascending ordered files) than the recursive version.

Based on only the comparisons and swaps, this sort seems best suited for ascending ordered files.

Efficiency: Time and Space

In general, we know that the sorts used in this lab have the following time and space complexities:

	Time			Space
	Best Case	Average Case	Worst Case	
QuickSort	$\Omega(n \log(n))$	$O(n \log(n))$	$O(n^2)$	$O(n \log(n))$
Insertion Sort	$\Omega(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Natural MergeSort	$\Omega(n)$	$O(n \log(n))$	$O(n \log(n))$	$O(1)$
Traditional MergeSort	$\Omega(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$

The quicksorts are reliant on their partition when it comes to best and worst-case complexity times. Median-of-three is going to usually produce average case time complexity, $O(n \log(n))$, because the partition is generated by the pivot, which is the median of the first, last, and center list values. Compare this to other quicksorts implemented in this lab where the pivot is the first value in the list. When the first value is a random number, the number of comparisons is closer to the number of comparisons when the pivot is the median-of-three. When the number is either the highest or lowest value in the lists, as in reverse and ascending lists, the number of comparisons is uniform, yet substantially higher than the number of comparisons for the same order of a file using median-of-three as the pivot. The quicksorts using the first value for the pivot more than likely produce time complexities resembling a worst-case scenario ($O(n^2)$). In the case where a random number is selected for the pivot, the time complexity may be closer to the average case ($O(n \log(n))$ for the initial quicksort only. The quicksorts employing the stopping cases also rely on insertion sort. Because these sorts are using the first list value as the pivot, producing unbalanced partitions, and using insertion sort (average time complexity of $O(n^2)$), the time complexity is more than likely closer to $O(n^2)$.

Here is a summary of the quicksorts at 10K file sizes:

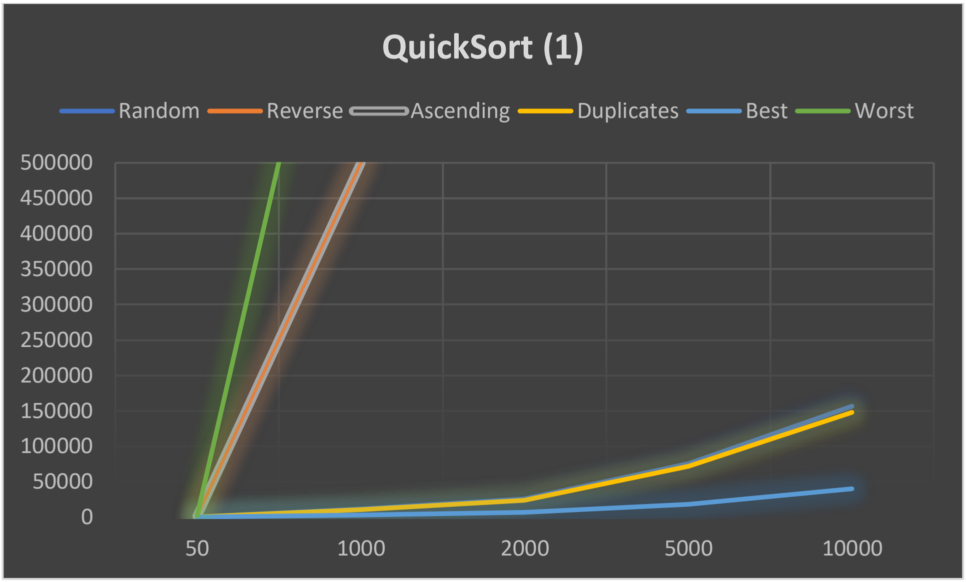
	QuickSort (1)		QuickSort Stop 100 (2)		QuickSort Stop 50 (3)		QuickSort Median (4)	
	Comps	Swaps	Comps	Swaps	Comps	Swaps	Comps	Swaps
Random	156574	29658	129325977	11300578	130866054	11299302	53797	43728
Reverse	49995000	5000	198955053	49975004	199458778	49975004	34519	16810
Ascending	49995000	0	148980050	0	149483775	0	29520	11808
Duplicates	147865	30022	135145920	13870804	138168270	13870804	54449	43743

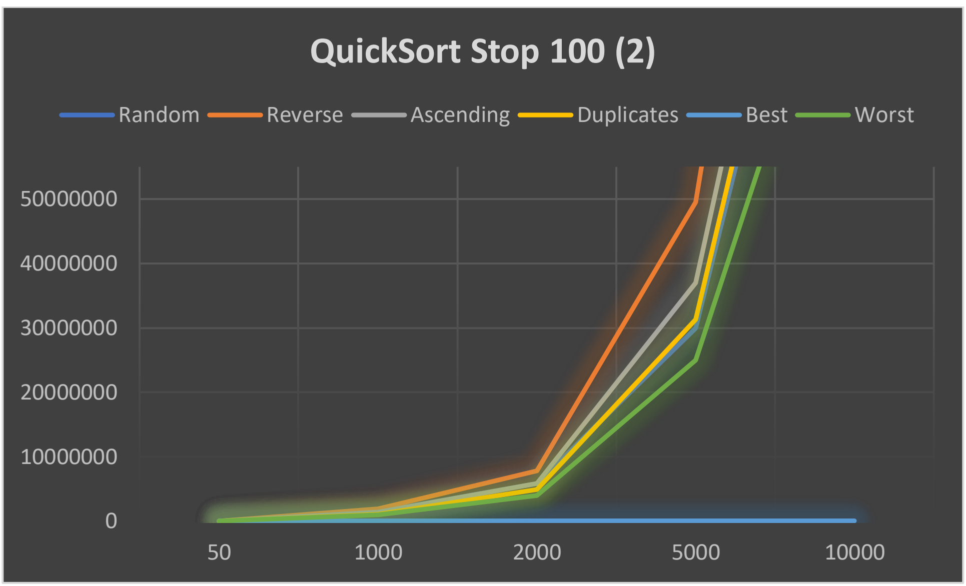
We observe that the number of comparisons is much lower when the median-of-three is used for the pivot. The number of swaps is improved for the other quicksorts when the file is in ascending order.

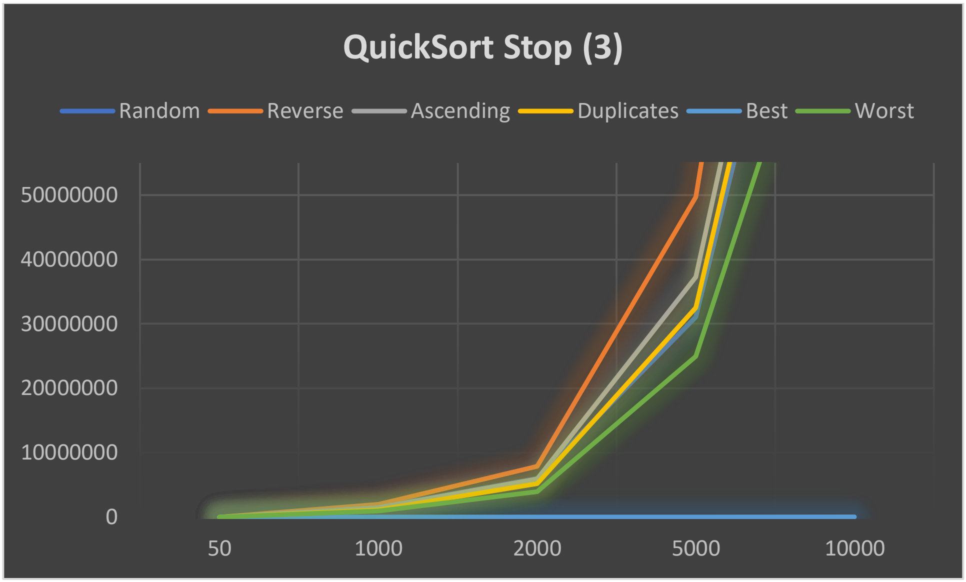
Based on the number of comparisons and swaps, as well as how we know the sorts are partitioned, I believe the time complexity would look something like this for the four quicksorts:

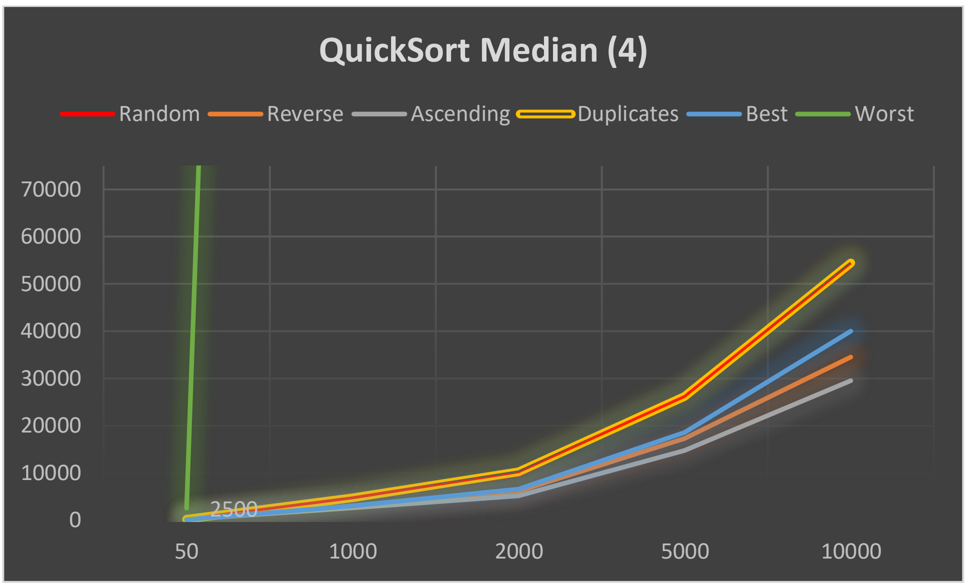
	QuickSort (1)	QuickSort Stop 100 (2)	QuickSort Stop 50 (3)	QuickSort Median (4)
Random	$O(n \log n)$	$O(n^2)$	$O(n^2)$	$O(n \log n)$
Reverse	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log n)$
Ascending	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log n)$
Duplicates	$O(n \log n)$	$O(n^2)$	$O(n^2)$	$O(n \log n)$

The corresponding graphs align with this assumption. I have plotted the best- and worst-case time complexities along with the number of comparisons for each sort in each order. The plots align with the assumptions made on the chart above.









Much like quicksort, mergesort relies on the partitions to sort a list through a series of comparisons. With the recursive Natural MergeSort (5), the method searches for runs of numbers in order and merges the runs together (also in order). The iterative Natural MergeSort with Array (7) should function the same way. Because it is iterative, the cost should be lower. The space complexity should be O(1) compared to O(n) by using the linked list instead of the array. The main advantage of the Natural MergeSort should be seen when processing input files in ascending order. We see this as the cost aligns with the expected best-case scenario (O(n)). Processing the random and reverse order files, as well as the files with duplicate values, provides inconsistent results. A worst case scenario of O(n logn) is expected, and the results seem to range from $O(n^2 / 6)$ and $O(n^2)$ to $O(2^n)$, respectively.

The Traditional MergeSort (6) partitions the list down the middle and continues partitioning until the list consists of individual values. The list is then sorted through a series of comparisons and merges. The expected time complexity is O(n logn), which was consistent when sorting reverse and ascending ordered files. Randomly ordered files and files with duplicates had a time complexity nearing $O(n^2)$.

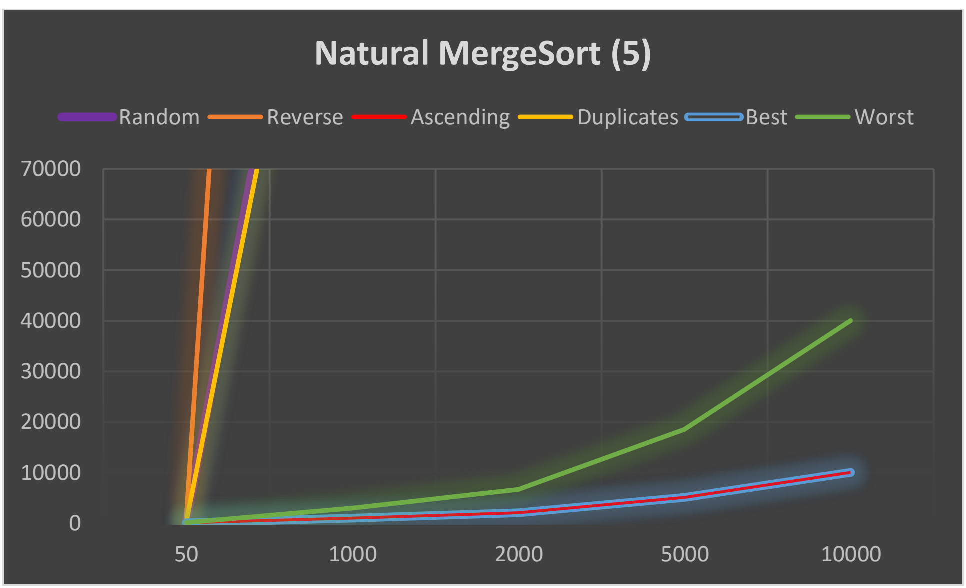
Here is a summary of the mergesorts at 10K file sizes:

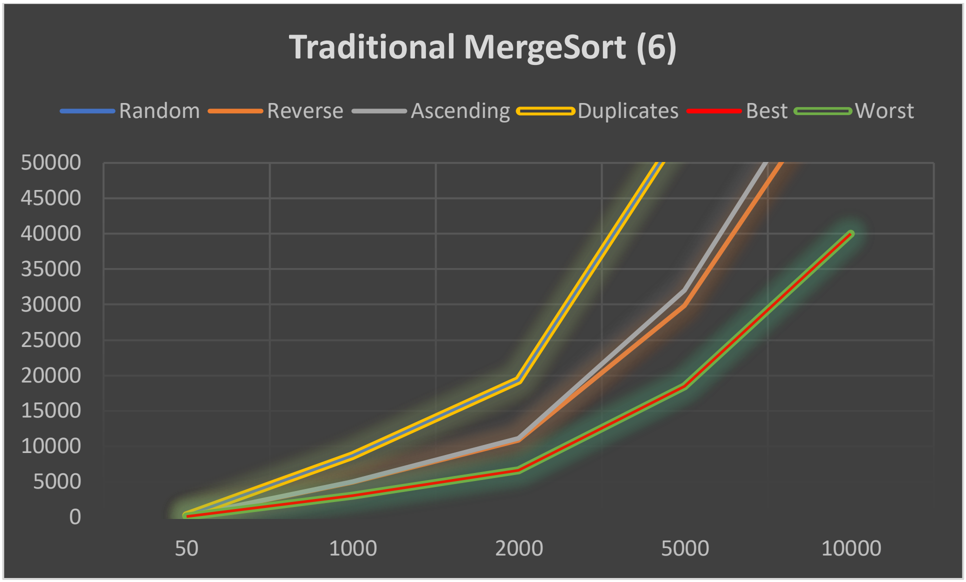
	Natural MergeSort (5)		Traditional MergeSort (6)		Natural MergeSort w Array (7)	
	Comps	Swaps	Comps	Swaps	Comps	Swaps
Random	16691319	16681330	120483	59112	128866	56009
Reverse	NA	NA	64608	64608	74607	64608
Ascending	9999	0	69008	0	9999	0
Duplicates	16507667	16497677	120413	59077	128738	55877

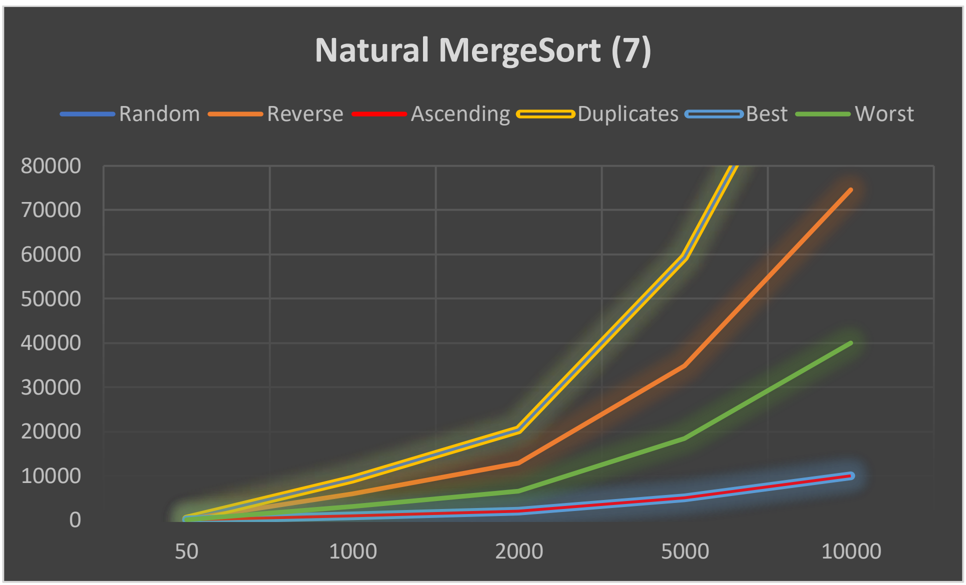
Based on the number of comparisons and swaps, as well as how we know the sorts are partitioned, I believe the time complexity would look something this for the three mergesorts:

	Natural MergeSort (5)	Traditional MergeSort (6)	Natural MergeSort w Array (7)
Random	$O(n^2 / 6)$	$O(n^2)$	$O(n^2)$
Reverse	$O(n^2)$ - $O(2^n)$	O(n logn)	O(2n)
Ascending	O(n)	O(n logn)	O(n)
Duplicates	$O(n^2 / 6)$	$O(n^2)$	$O(n^2)$

The corresponding graphs align with this assumption. I have plotted the best- and worst-case time complexities along with the number of comparisons for each sort in each order. The plots align with the assumptions made on the chart above.







We were instructed to use linked implementations for the mergesort. I implemented both for this project. After doing some research, I found that that the linked implementation would provide a O(1) space efficiency and the array implementation would provide a space efficiency of O(n). The seems to make a larger difference as the size of the input files increase. The higher space efficiency seems to be the result of using arrays to temporarily store the values of the list to be sorted (or merged in this case). In the linked implementation we are not temporarily storing, so we avoid that cost.

To summarize, the method used to partition the input, as well as the order of the input, appear to be the most important factors in terms of efficiency. We can see this as we compare the sorts. A file provided in ascending order can most efficiently be processed by a Natural MergeSort because it takes advantage of the order of the file. It only compares the number of items in the file – as opposed to additional comparisons needed to merge or sort in other sorts. If I was looking for an all-purpose sort because I might receive files in any order, I would probably select a quicksort with a median-of-three as a pivot. This ensures that I am generating partitions that can efficiently process any size file. The results of my tests confirm that this is the case.