Andrew Taylor

Atayl136

Lab 4 Analysis: Sorting Complexity using Quicksort and Natural Merge

Introduction

I was really happy, very glad to do this lab. This is the kind of knowledge I had to see first hand and I'm sure I will apply often down the road in my studies and career. The task seemed straight forward, but there were challenges along the way that I'm glad I encountered here, first.

This is an exercise I should really conduct for each and every sort and search I consider in my programming career. The information is invaluable. The insight into complexity it provides is needed, to make sound decisions and make data problems tractable. I'm not that good at understanding complexity I feel, because Big O seems so amorphous. Are all O(n lg n) the same? After doing this lab I think the answer is "No".

The Data Structure

The data itself came provided with the lab, and I used all the required files. Numbers were space-delimited and contained no labels or dataframe, so it was easy to read in and store in memory. One thing I didn't do, (among others) is read the input one character at a time. I may lose points for this but on this lab it didn't seem to make sense. There was no consideration of bad data, and that would change my mind if there was. The complication of breaking numbers up into digits and retabulating them with a complex logic doesn't seem practical.

The numbers were stored in lists, a simple and effective way to structure the sorts. Lists are my favorite data type in Python. I find them flexible yet powerful, suited for a wide variety of applications. I wonder how the figures would have turned out using NumPy arrays. One day I should look into this.

Recursive versus Iterative Sorting

Recursive solutions are a popular choice for solving problems that can be broken down into smaller subproblems, especially those with a divide-and-conquer structure. This is because recursion utilizes the (memory) call stack to efficiently decompose complex problems into more manageable subproblems. My recursive versions of the sorts are good examples of this approach. I didn't understand the instructions wording of "one or two" because a partition of two elements could still be unsorted unless the algorithm completes. So I made the stopping case when high index and low index meet.

However, when it comes to processing large files, recursion may not be the best option. This is because recursive functions call themselves repeatedly, which can result in a stack overflow when the call stack exceeds its memory limit. I encountered this problem when I tried to process files larger than 2k entries. This limitation makes iterative solutions a better choice for processing large files. It was actually frustrating because I encountered this problem only after coding all the recursive functions. To fix this error, would have to reduce the recursive calls in the code (which I couldn't think how), use iteration instead of recursion, or increase the stack size limit. Increasing the stack size limit can be done using tools like ulimit or the Windows registry editor, but I was not going to tinker with my Windows just to complete the assignment. The message was clear, recursion is NOT suitable for sorting files of this

size. So I had to do everything over iteratively. I suggest that the Lab instructions might be corrected, so as not to frustrate students, but perhaps that was the lesson to learn. So, I learned the hard way, I did both.

Although iterative solutions may require additional data structures like stacks or queues to manage state, they can be more memory-efficient, especially in cases like these. For the quicksorts and natural merge sort, if the file size is small I prefer the recursive approach because it aligns well with the natural division of the problem into smaller subproblems, and also, the code is more elegant. The recursive function is called on each of these subproblems, continuing until the base case is reached. This recursive partitioning yields an efficient sorting process with a time complexity of O(n*log(n)). An iterative implementation of these algorithms would require an explicit stack to track subarrays in need of sorting, making it more difficult to implement and understand. But the time complexity in theory should be comparable, however, the space complexity is not. This was the issue of recursion exceeding RAM.

In conclusion, while recursion is an elegant and intuitive approach for solving some types of problems, it may not be suitable for processing large files due to the risk of memory stack overflow. Iterative solutions, while they may be more memory-efficient, can be more challenging to implement and comprehend. I'm just pleased I got something to work for all files provided.

Design

I'm not a polymath and I didn't invent these classic sorts, so I referred to online resources to find the right code for the job. I had to adapt what was indicated for the tracking of comparisons and exchanges, but that wasn't too much work. It was as was said in lecture, this are time-honored methods that are readily available and used over and over in computer science. I felt it wasn't up to me to redesign what is taken as established. Other than these changes much of the code was as I found it.

The process function was simple, I revised it until the output conformed to the requirements, and made sure to generate an extra file that summarized all the sorts in one. I thought about timing my functions as I did in Lab 1, but decided against it because Scott Almes said to focus on comparisons and exchanges; that runtime is particular to computers and computer configurations. So that's what I did.

Efficiency

I'd like to start this section with a great table I refer to over at Geeks for Geeks. This shows the time and space complexity of popular sorts. Perhaps you've seen it before:

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Bucket Sort	Ω(n +k)	θ(n +k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n + k)
Count Sort	Ω(n +k)	θ(n +k)	O(n +k)	O(k)
Shell Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(1)
Tim Sort	Ω(n)	θ(n log(n))	O(n log (n))	O(n)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Cube Sort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

As far as the two types of sorts in this lab are concerned, they both seemed to be of the complexity O(n log n) with one key difference: Quicksort has a quadratic worst case. This really showed up in the numbers. At the end of this report I include my charts of performance for all sorts grouped file by file. These bar charts provide a fast way to see the differences between each sort on all the proving grounds: numbers in ascending, random, and reverse order in the sizes of 50, 1000, 2000, 5000, and 10000. One note, I did use the provided files, and was grateful for them, but I think that the random .dat files have more than 1% duplicates…it seems like 100%. But, I figured this was intended so I didn't change them.

From a bird's eye, it was remarkable how in almost all cases there were clear winners and losers for each category and file size. Sorts performed well in some cases, but not in others. This highlighted the need for me as a practitioner to be knowledgeable and mindful of sort selection in future applications.Although charts are provided, let's take a tabular view here to frame the discussion:

	rev5K.dat	4999	12497400	12497450	70063	51822	12497499
	rev50.dat	49	1225	1225	213	193	1224
	rev2K.dat	1999	1998900	1998950	25737	17964	1998999
	rev1K.dat	999	499400	499450	12252	7987	499499
	rev10K.dat	9999	49994900	49994950	153088	113631	49994999
	ran5K.dat	4141061	128737	95833	69644	62898	76667
	ran50.dat	499	1857	1014	670	520	560
	ran2K.dat	668176	47314	33273	24208	21253	25090
	ran1K.dat	160970	24030	15049	10728	9434	11244
	ran10K.dat	16510068	268210	195234	162159	129261	154933
	asc5K.dat	0	12492450	12496225	70459	51822	12497499
	asc50.dat	0	0	0	230	193	1224
	asc2K.dat	0	1993950	1997725	24585	17964	1998999
	asc1K.dat	0	494450	498225	11170	7987	499499
	asc10K.dat	0	49989950	49993725	150765	113631	49994999
Comparisons	Sorting Algorithm	Natural Merge	Quicksort 100	Quicksort 50	Random Pivot Quicksort	Median Quicksort	Quicksort

	rev5K.dat	9998	14746	11121	44774	6355	4999
	rev50.dat	98	1225	1225	186	66	49
	rev2K.dat	3998	8746	5121	14916	2067	1999
	rev1K.dat	1998	6746	3121	7161	1031	999
	rev10K.dat	19998	24746	21121	95658	12711	9999
	ran5K.dat	9990	86577	48387	44865	14597	13871
	ran50.dat	194	1857	894	458	158	148
	ran2K.dat	3991	35245	19371	13381	5207	5082
	ran1K.dat	1991	19309	9065	6788	2361	2261
	ran10K.dat	19987	183428	99590	86141	31969	30502
	asc5K.dat	4999	9797	9897	43616	3857	4998
	asc50.dat	49	0	0	180	43	48
	asc2K.dat	1999	3797	3897	15445	1069	1998
	asc1K.dat	999	1797	1897	7776	533	998
	asc10K.dat	9999	19797	19897	88658	7713	9998

Exchanges	Sorting Algorithm	Natural Merge	Quicksort 100	Quicksort 50	Random Pivot Quicksort	Median Quicksort	Quicksort

Some things made sense immediately, with the understanding of how these algorithms work. Natural merge, being an optimization of merge sort that identifies pre-sorted areas called “runs” in the input data, performed extremely well in comparison of ascending, ordered data. However, the exchanges were extreme, leading me to believe I had not coded this sort correctly. Shouldn’t there be as many comparisons in identifying the run, and no exchanges considering it’s in order? Maybe I coded it wrong, I hope you can let me know in the feedback. Natural merge deviates from the table and sorts ascending and descending runs in O(n) time.

Being O(n), and Quicksort topping out at O(n^2), Natural Merge won in several categories: comparisons for reversely ordered data, exchanges for ascending data. It also performed well in random exchanges, but very poorly in random comparisons. This highlights the wisdom of keeping personal records on these figures. I’d like to put this in my Github and round out the exercise over time so I have a resources because the nuances are important to keep track of.

The many of flavors of Quicksort tested, including the one I added with a random pivot, exhibited a great variety of performance metrics. Interestingly, plain vanilla Quicksort was clear winner across the board on Exchanges, but the number of Comparisons required made the algorithm inefficient. That was a point of clarity for me. Quicksort50 and Quicksort100 were the Quicksorts completed by an insertion sort in the last 50 and 100 entries, respectively. This was interesting, because in the case of size 50 files the statistics were essentially measuring the performance of insertion sort. I didn’t count the comparisons and exchanges of insertion sort however, because it wasn’t a factor in all sizes of files, and that wasn’t the assignment I felt. Perhaps I should have, but presumably on that size of 50 and 100 time complexity gains would be negligible so I skipped it.

One thing that popped out was the degree to which a median-of-three or random pivot improved Quicksort. I mean, they said it was better in lecture and reading, but, what a difference! This is what leads me to believe that not all O(n log n)’s are the same, because although Big O ignores the coefficients they can make this kind of difference I see. Isn’t that what’s happening here?

These were some of the takeaways from the metrics tht I saw, and given these cases above there are particular tools for particular situations. If I had to choose one sort to proceed initially, without time for investigation of the dataset, I would choose median-of-three pivot Quicksort because it seems like the numbers come in towards the lower end, especially in comparison to plain vanilla Quicksort.

Overall, the largest determinant of performance was the size of the file, with reverse order coming a close second. For some reason ordering the file in reverse created better performance (generally) for all the sorts I studied. Another place I didn’t know how to make requirements was the graph of actual versus theoretical performance. I wasn’t sure how to do that so unfortunately it is omitted. I hope you can kind of see that from the bar charts and what was said here?

Enhancements

The major enhancement I included was an additional sort: The Random Pivot Quicksort. I’m glad I did. This was important to see. The gains from this approach are massive. This makes me want to flesh out the lab on my later on with all the sorts included in the Geeks for Geeks table.

Lessons Learned

All in all, I was able to complete the lab and get valuable information although there were a few places I came up short. Also, I didn't include the capability for named files because I already demonstrated that on a previous lab and it would be a pain to run the program if you had to put in each .dat file separately. I think I made choices that made sense. If I had to do the lab over again I would:

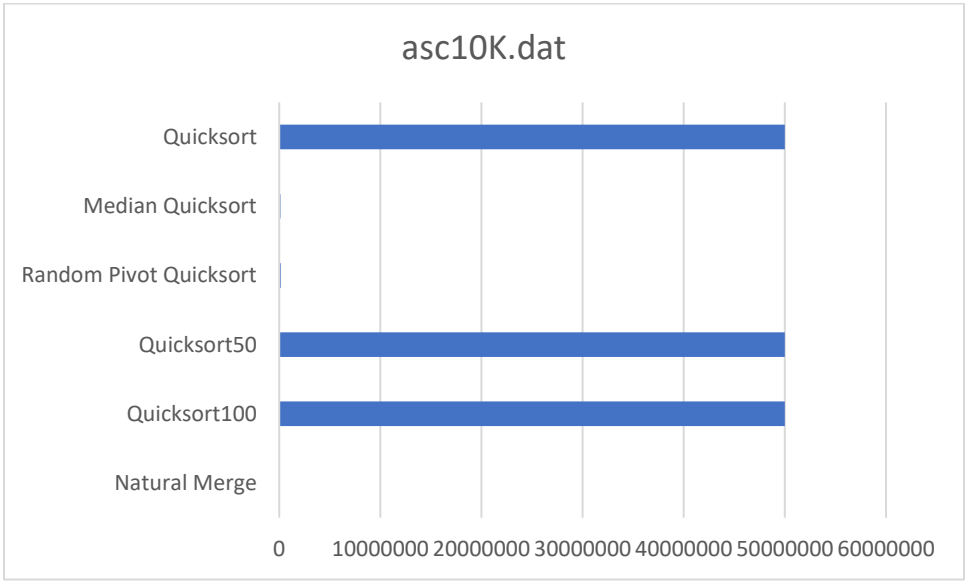
1) Skip the recursive approach.
   Although this appealed to me because recursive code has really grown on me, I found out that recursion is so inefficient for sorting that it couldn't be executed on my computer for file sizes above 2K. I think the lesson is that it's not appropriate for anything but small sorts, so I'll keep that in mind clearly when I program in the future.
2) Check the table and the times.
   I would like to have timed the functions but I ran out of time and couldn't figure out an easy way to do this without chunky code. I had done this on a prior lab, but I didn't do this here, and I regret not adding that enhancement.
3) Consider modularity, could I have used it?
   Although I think it makes sense to turn in two files here (with such a small process function), I could have made the program modular and somehow built the timing into a module. I would have done this if I didn't have to do the lab over iteratively.

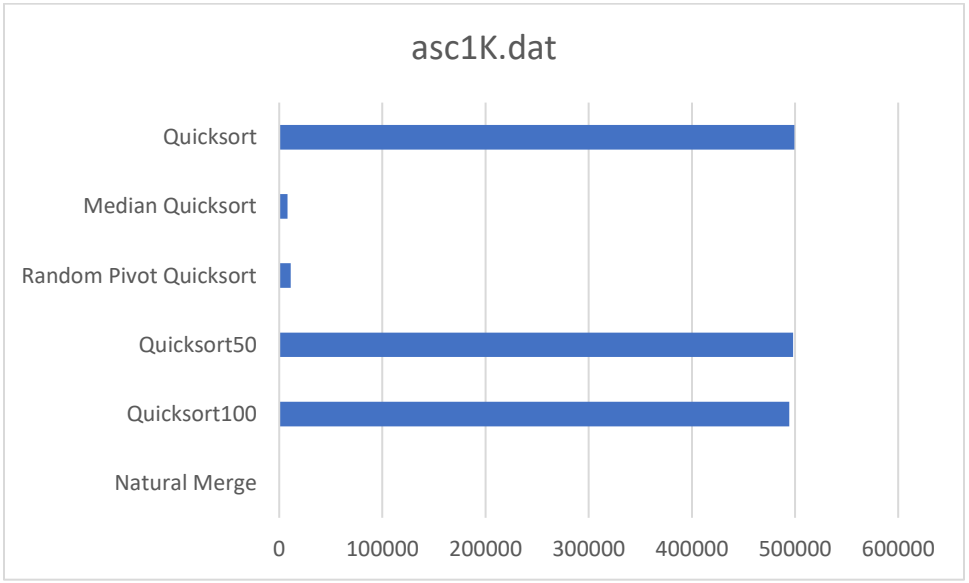
Conclusion

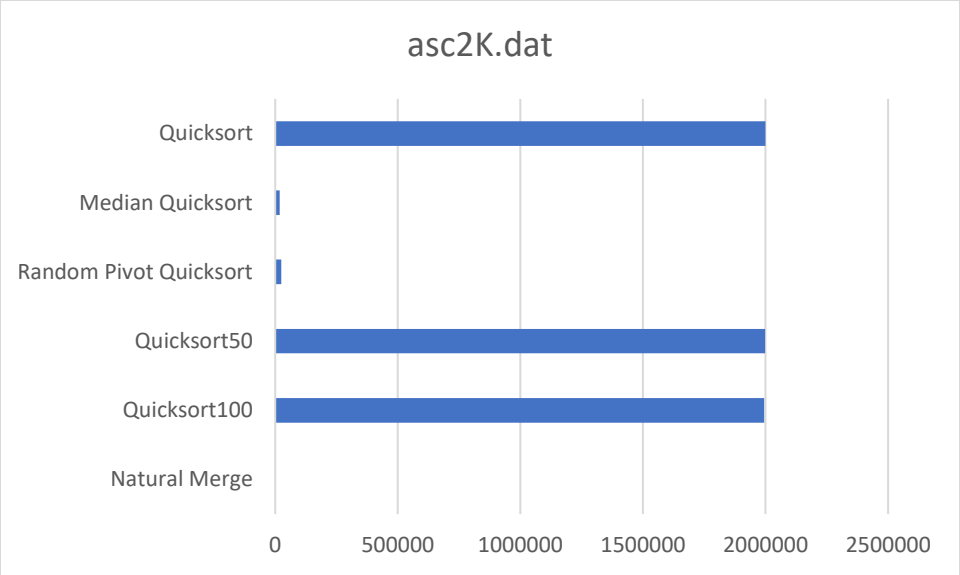
In conclusion, I'm very happy with the lab and with the class. Seeing this for myself has taught me to think strategically about data: how it is stored, structured, accessed, labeled, and understood. The skills I picked up in this lab will be used over and over, and deserve to be fleshed out, becoming a best practice for the rest of my career. It was well worth the time. Thank you!

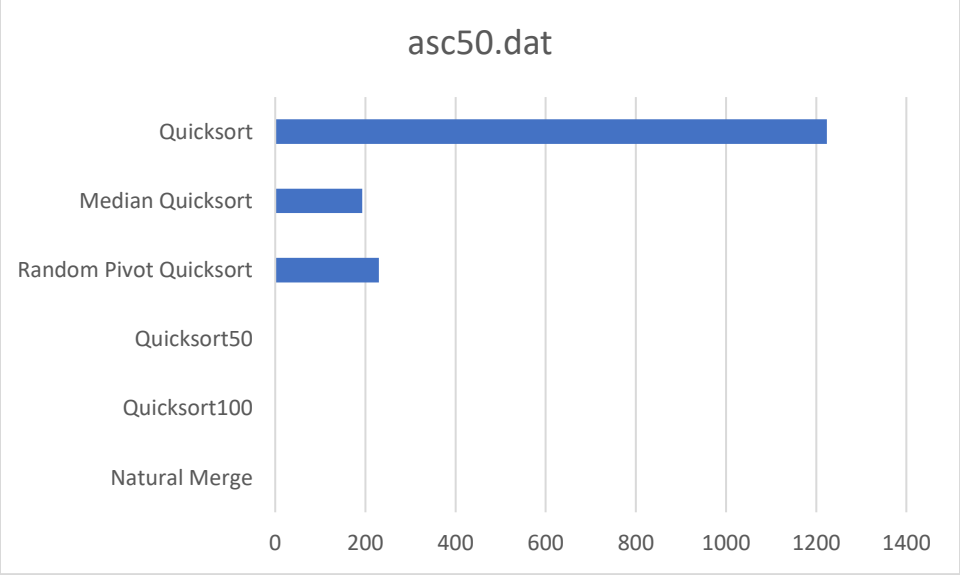
Contrasting Complexity in Bar Charts

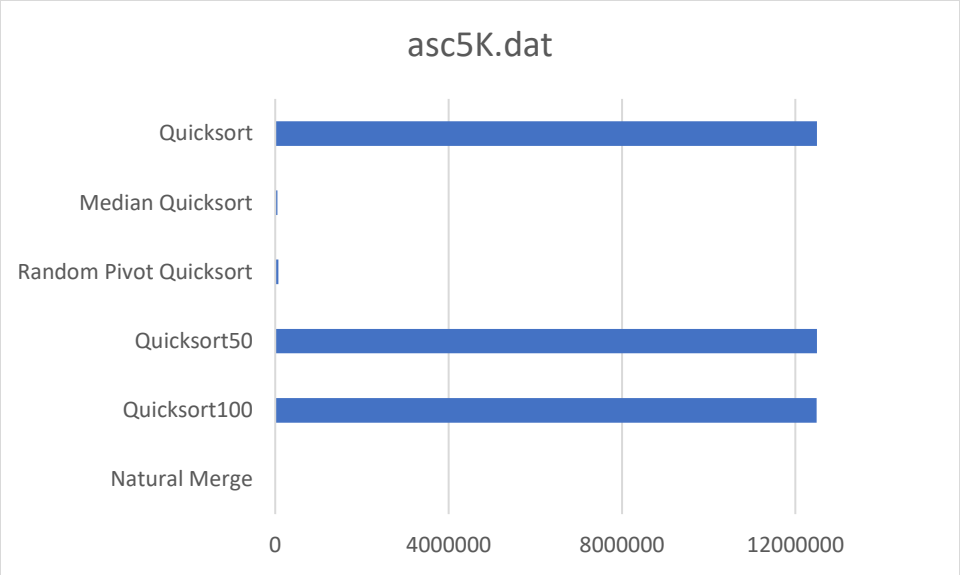
Comparisons Calculated per Sort, File by File:

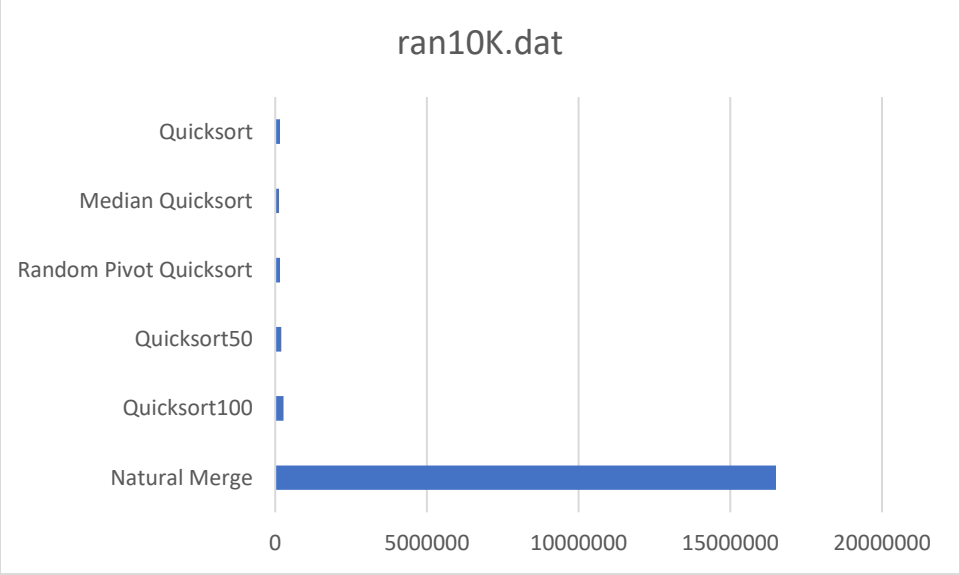


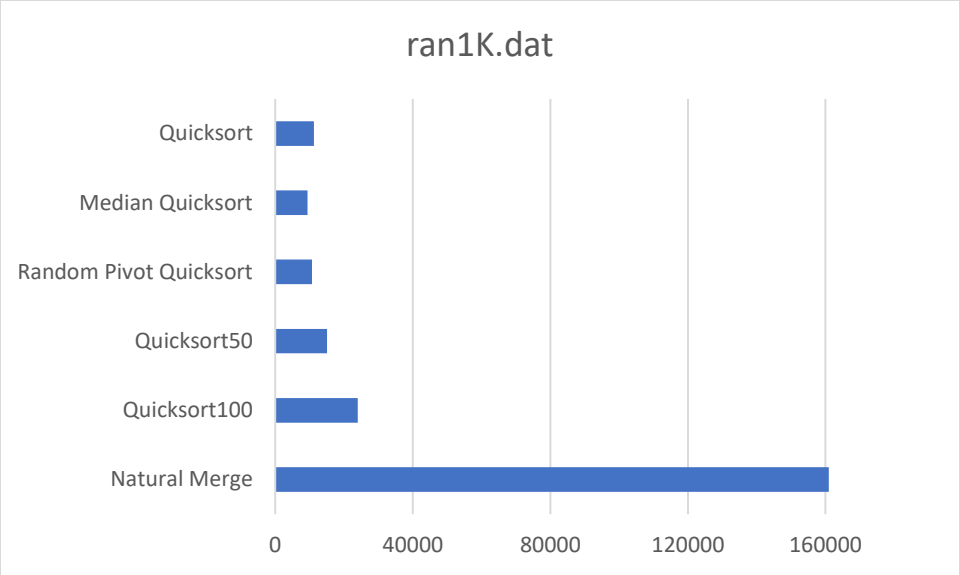


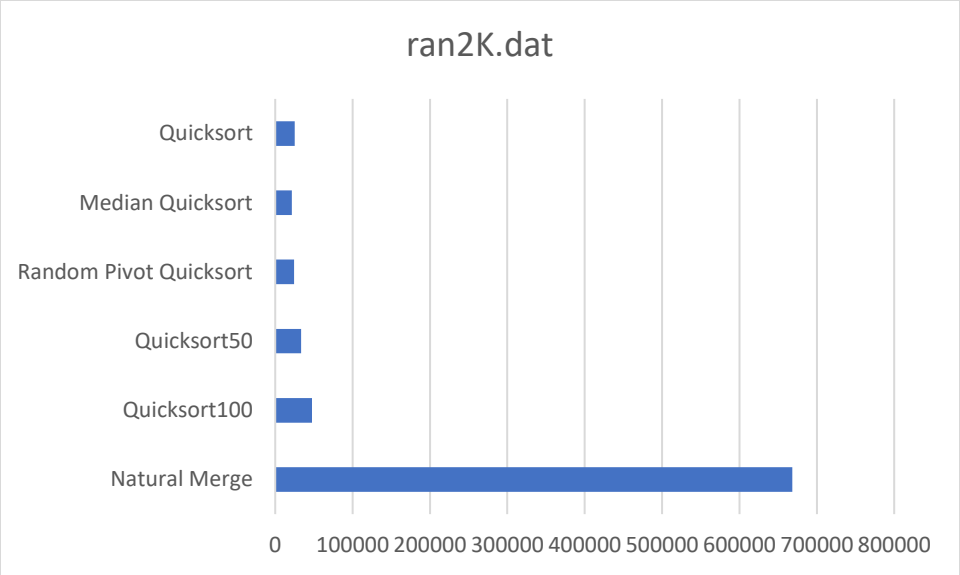

asc2K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
500000
1000000
1500000
2000000
2500000

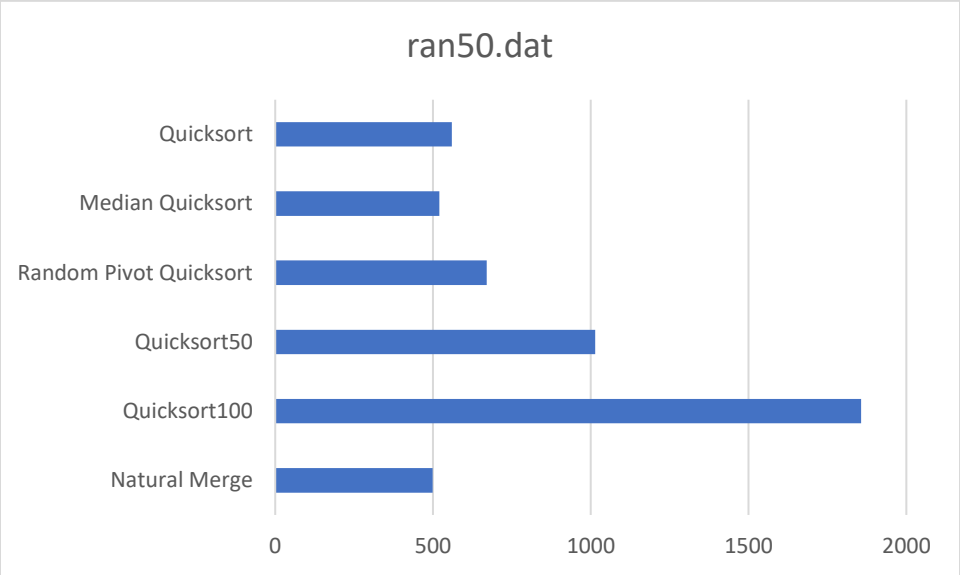

asc50.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
200
400
600
800
1000
1200
1400

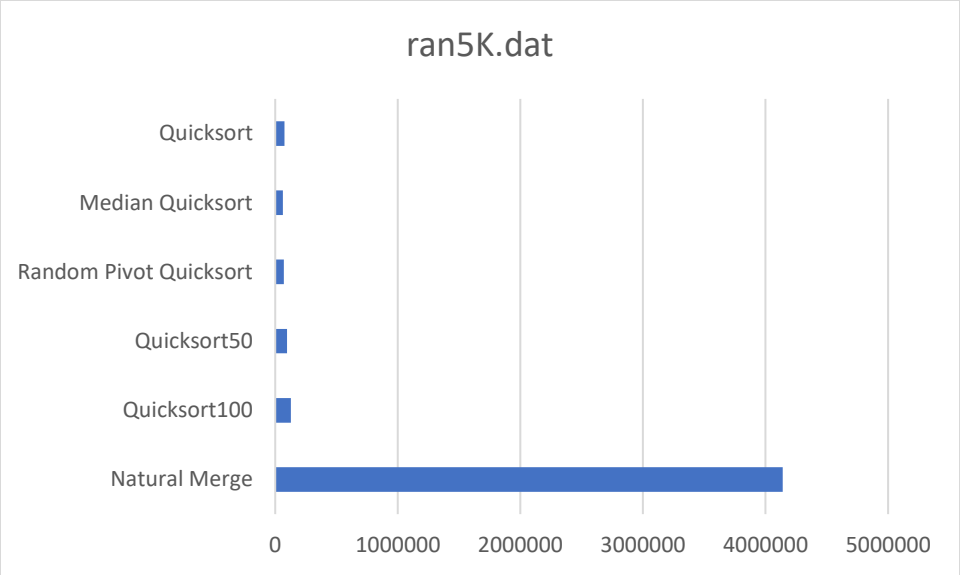

asc5K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
4000000
8000000
12000000

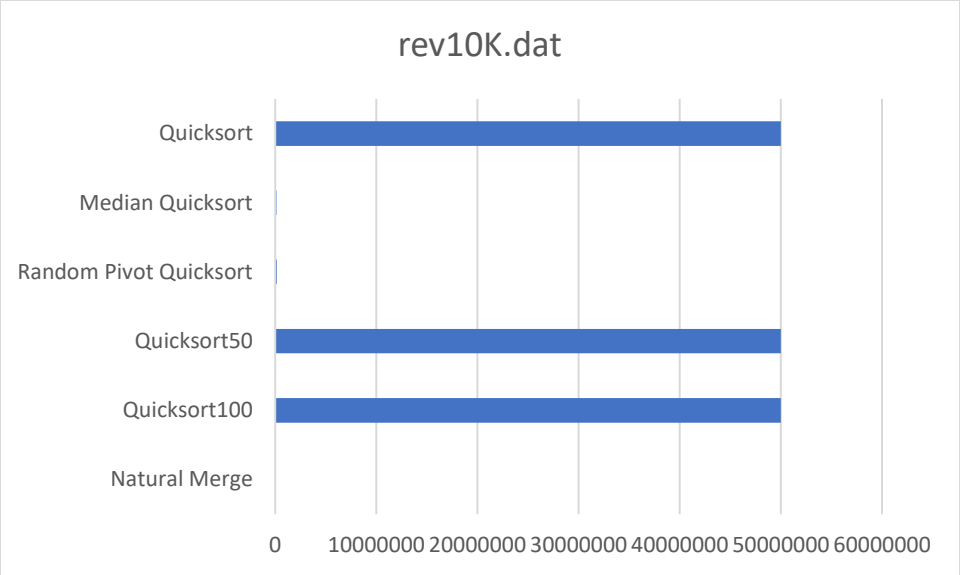

ran10K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
5000000
10000000
15000000
20000000

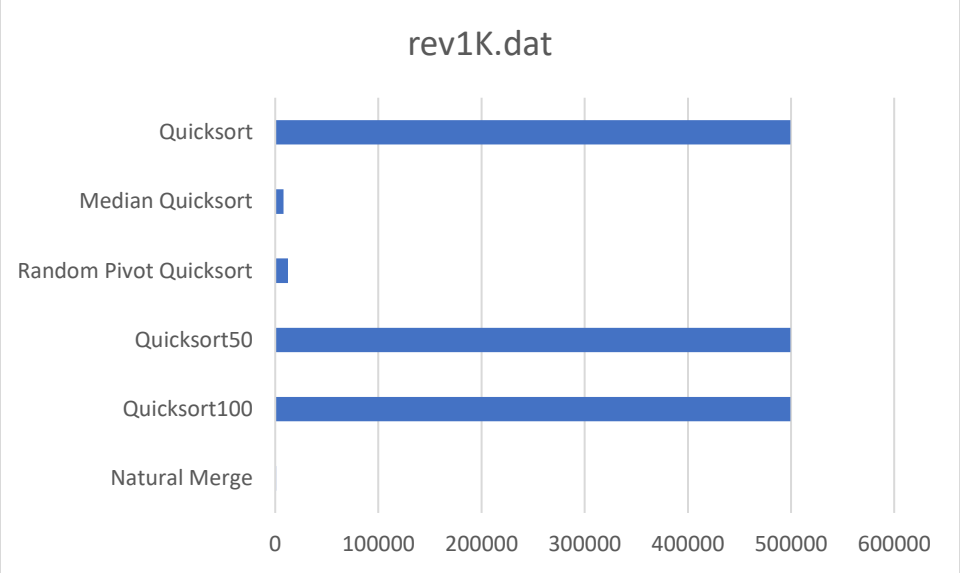

ran1K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
40000
80000
120000
160000

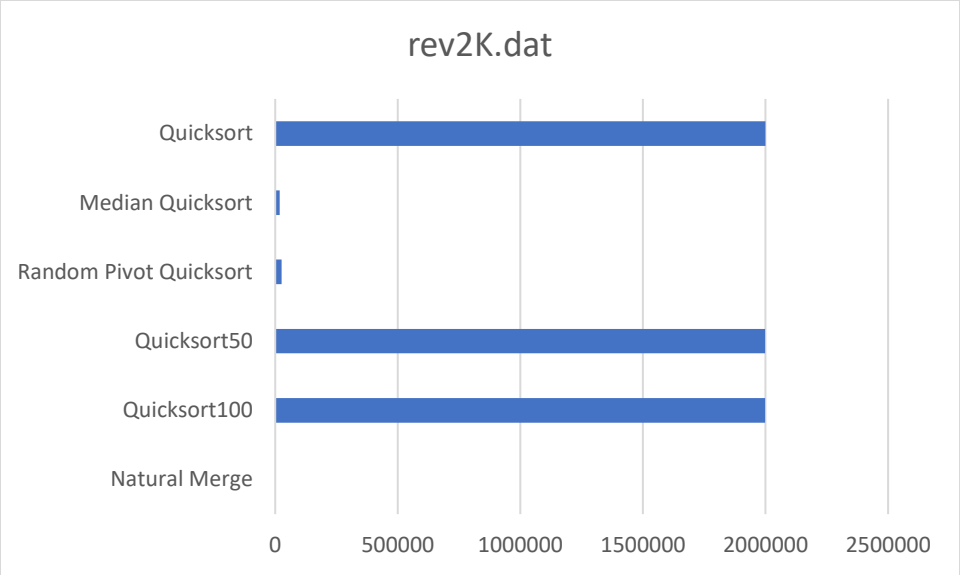

ran2K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
100000
200000
300000
400000
500000
600000
700000
800000

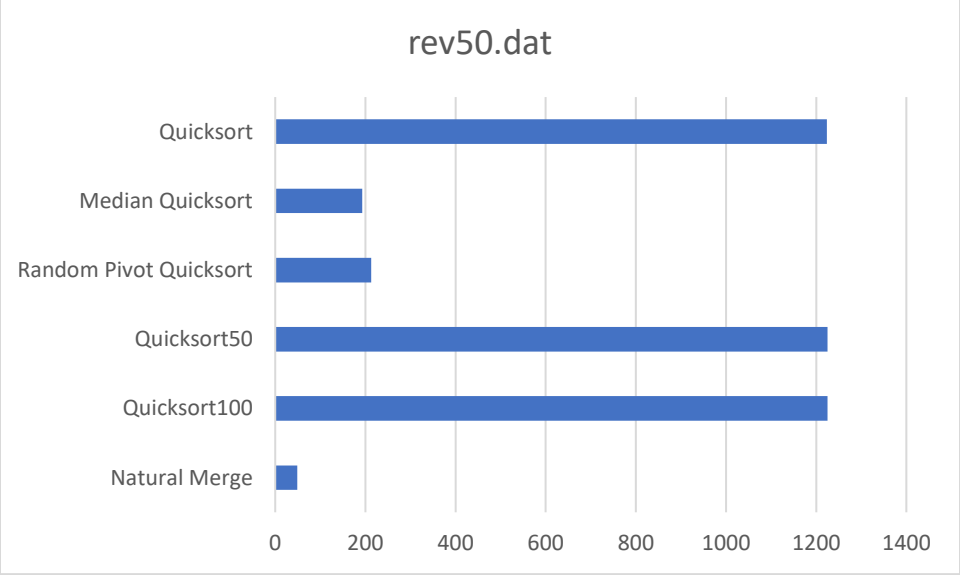

ran50.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
500
1000
1500
2000

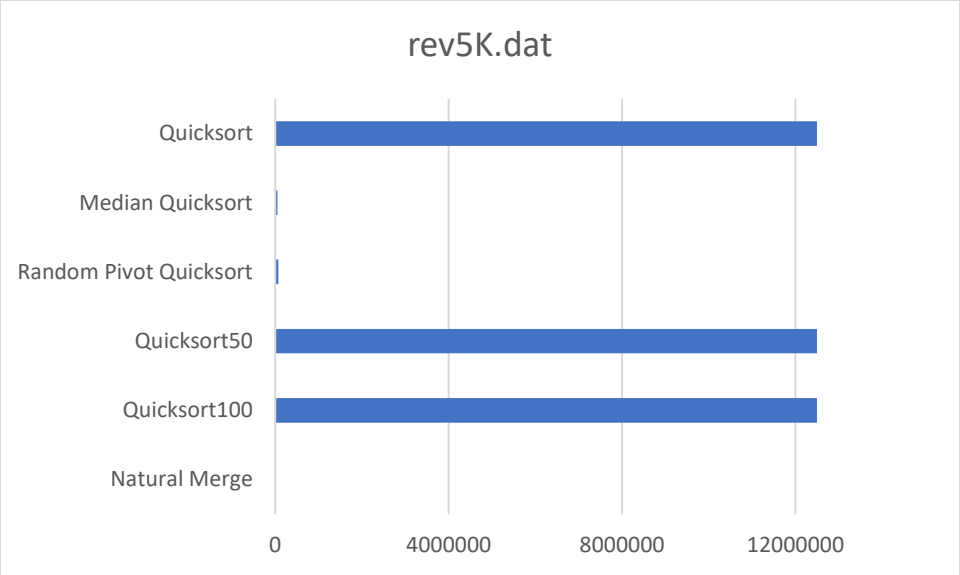

ran5K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
1000000
2000000
3000000
4000000
5000000


rev10K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
10000000
20000000
30000000
40000000
50000000
60000000

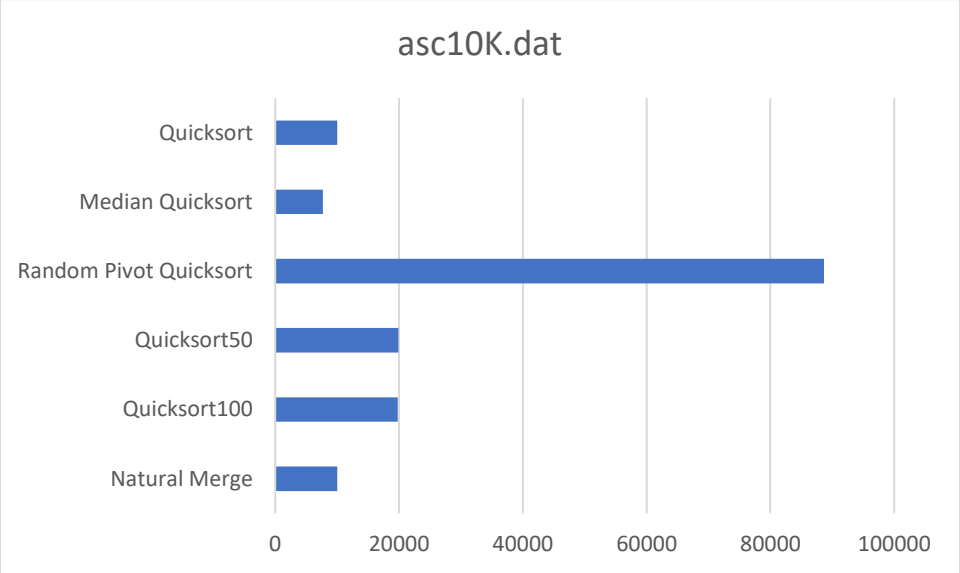

rev1K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
100000
200000
300000
400000
500000
600000

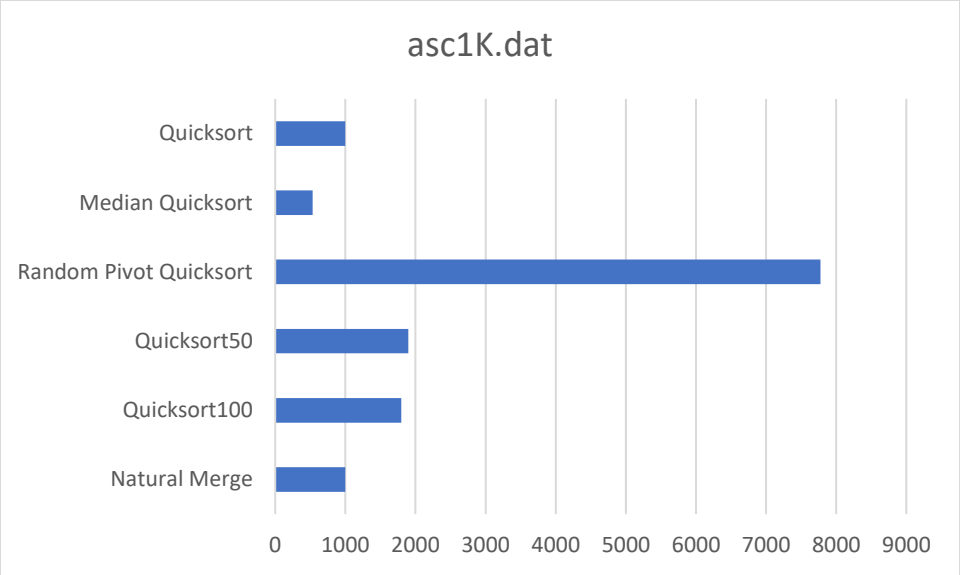

rev2K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
500000
1000000
1500000
2000000
2500000

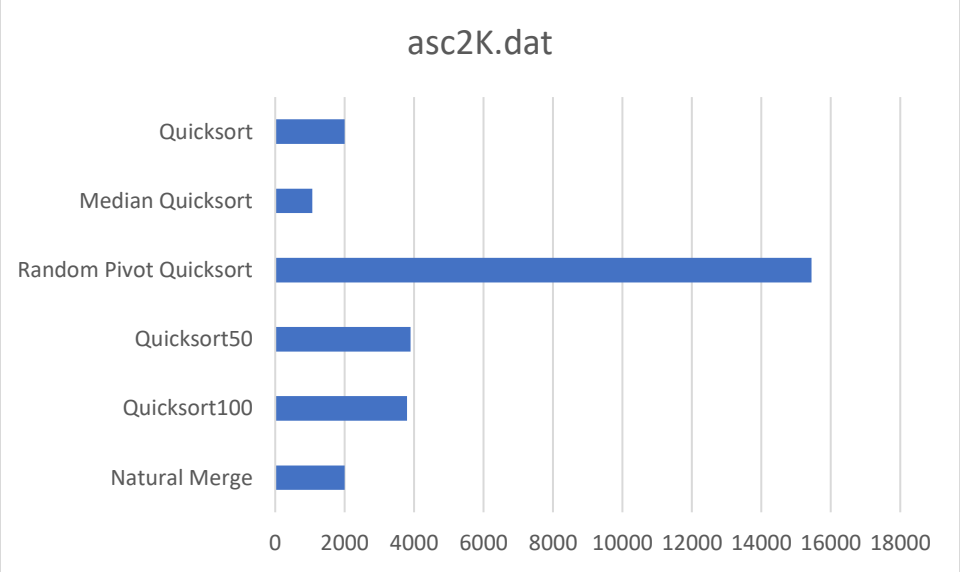

rev50.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
200
400
600
800
1000
1200
1400

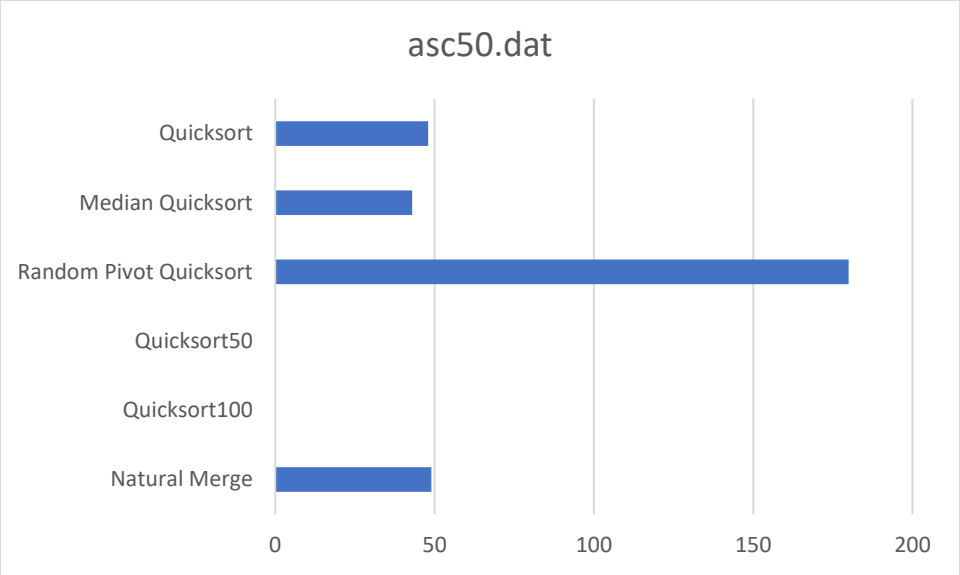


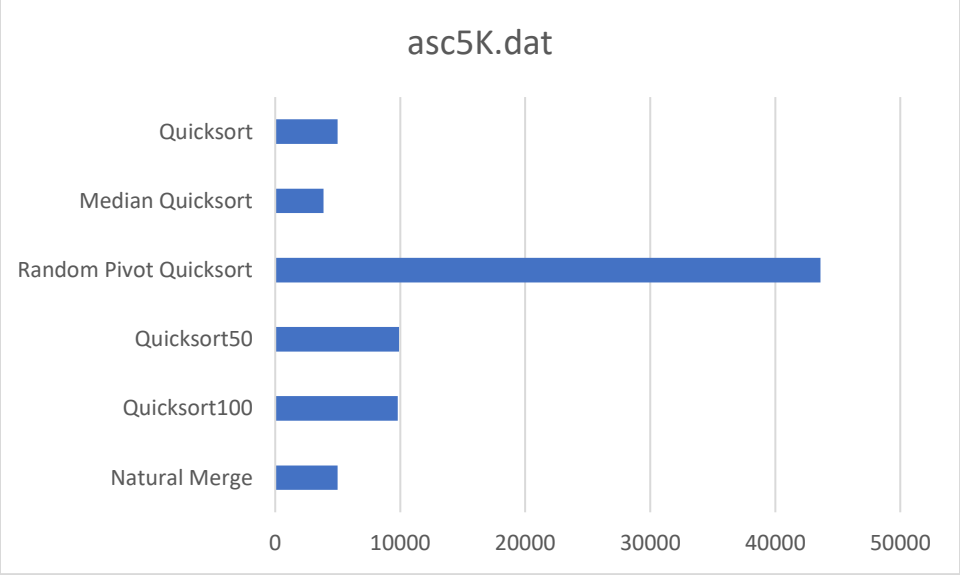
Exchanges per Sort, File by File:

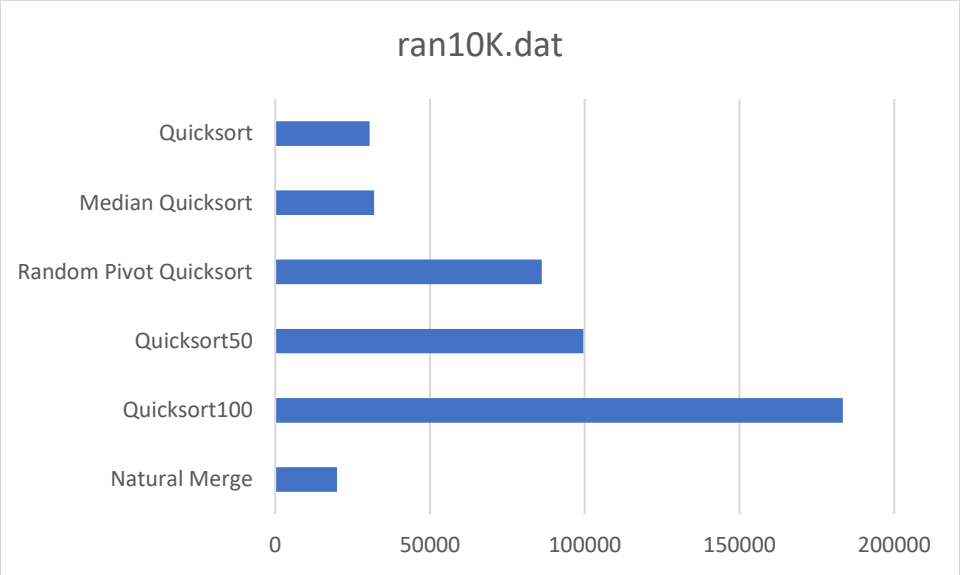


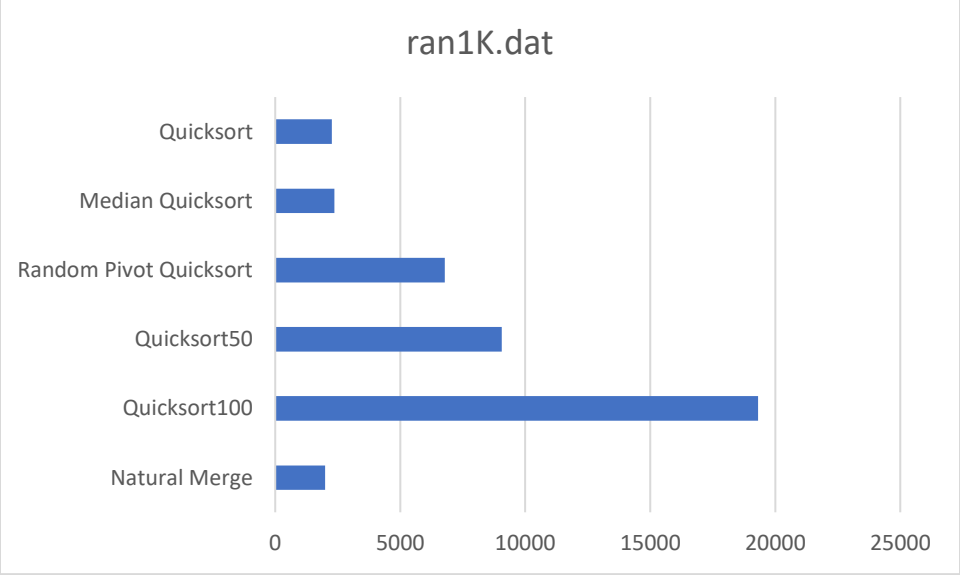

asc1K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
1000
2000
3000
4000
5000
6000
7000
8000
9000

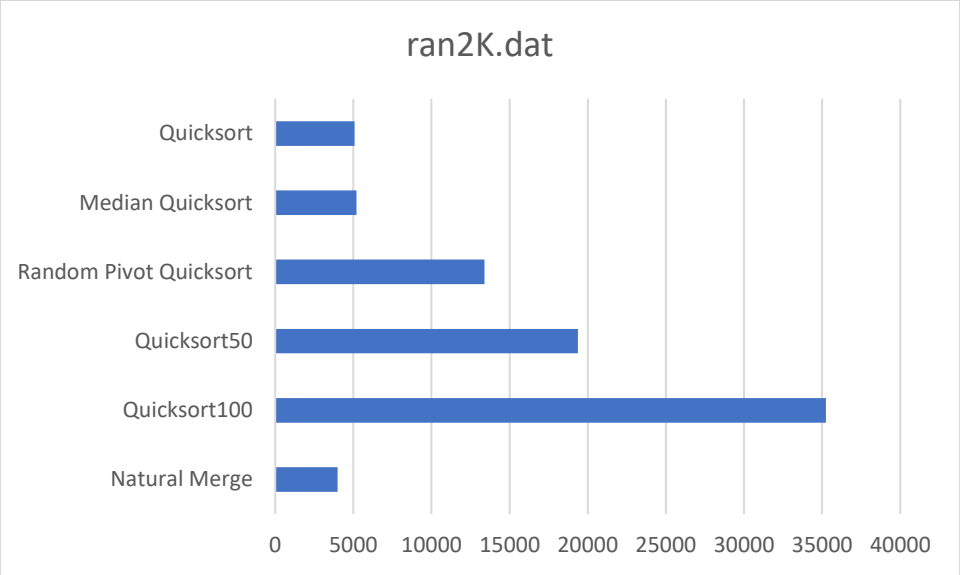

asc2K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
2000
4000
6000
8000
10000
12000
14000
16000
18000

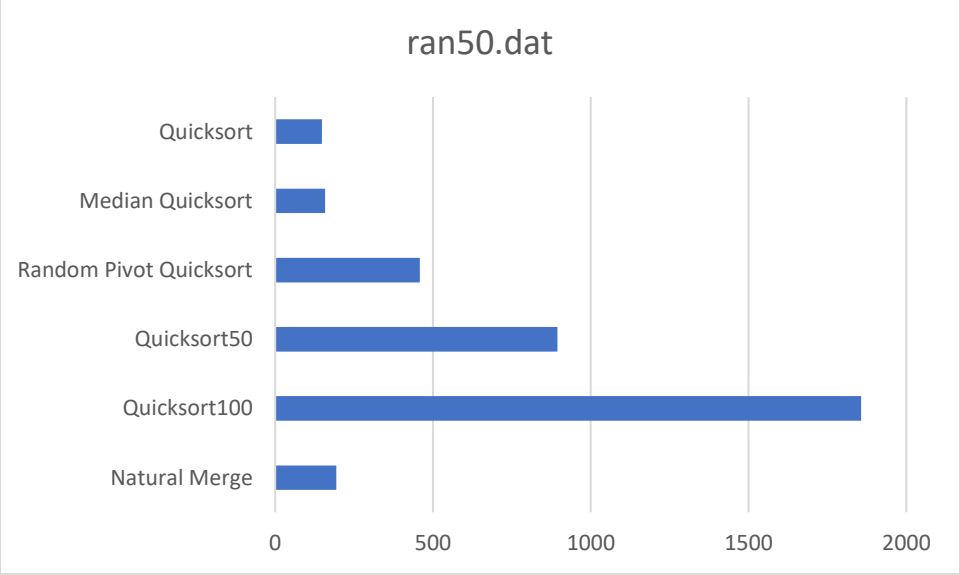

asc50.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
50
100
150
200

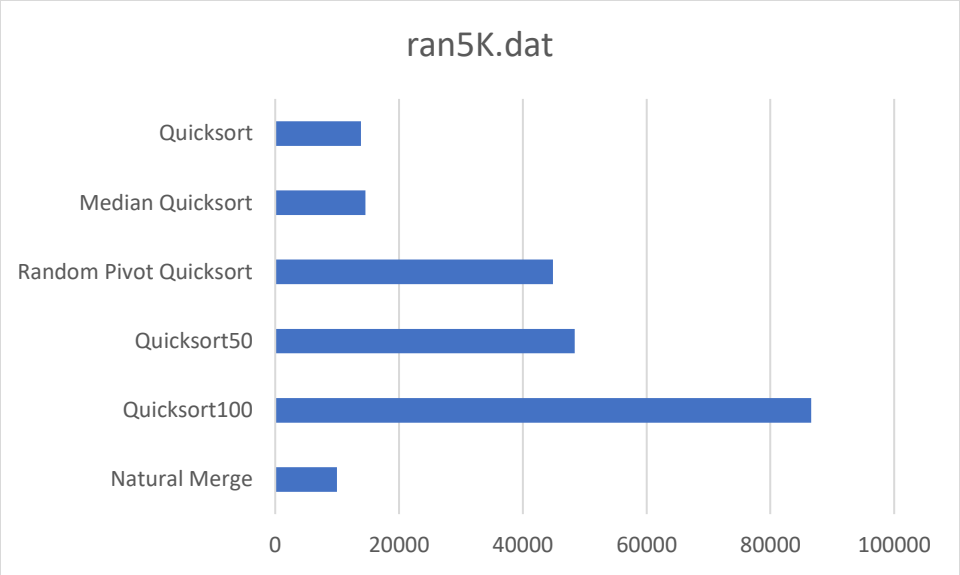

asc5K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
10000
20000
30000
40000
50000

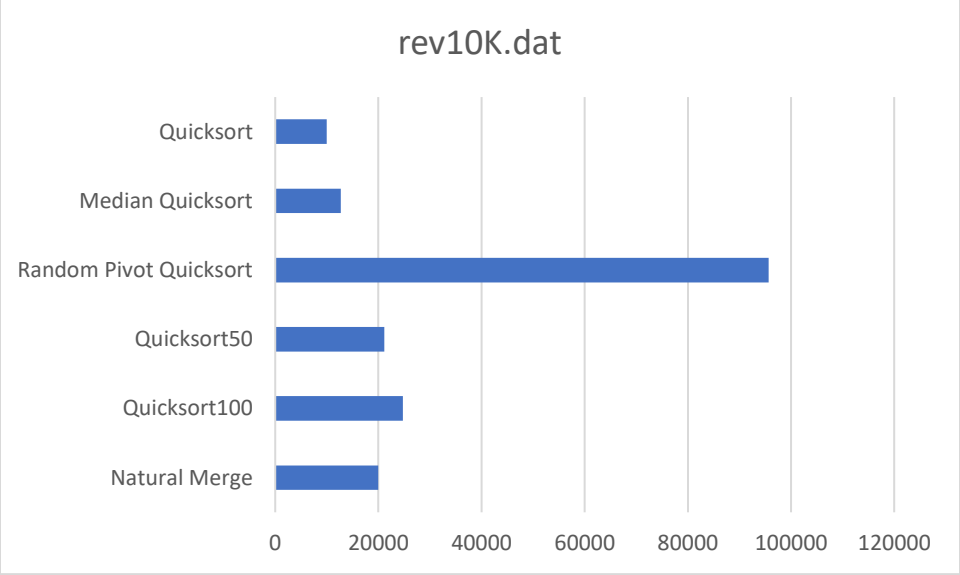

ran10K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
50000
100000
150000
200000

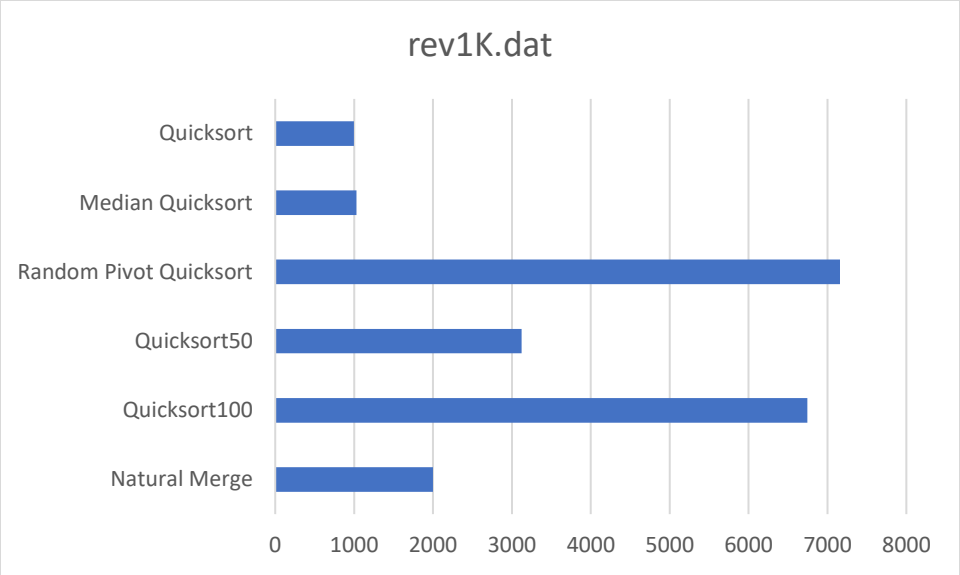

ran1K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
5000
10000
15000
20000
25000

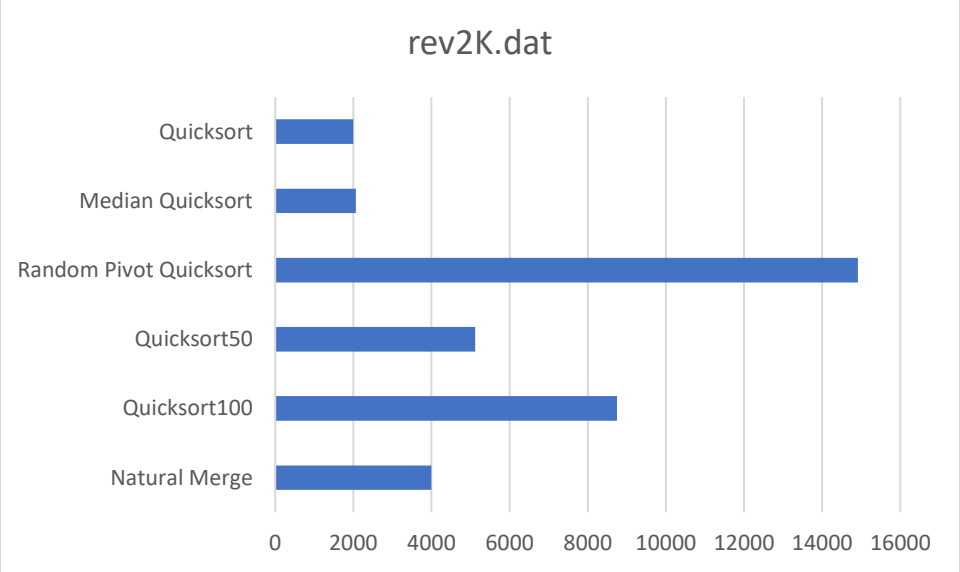

ran2K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
5000
10000
15000
20000
25000
30000
35000
40000

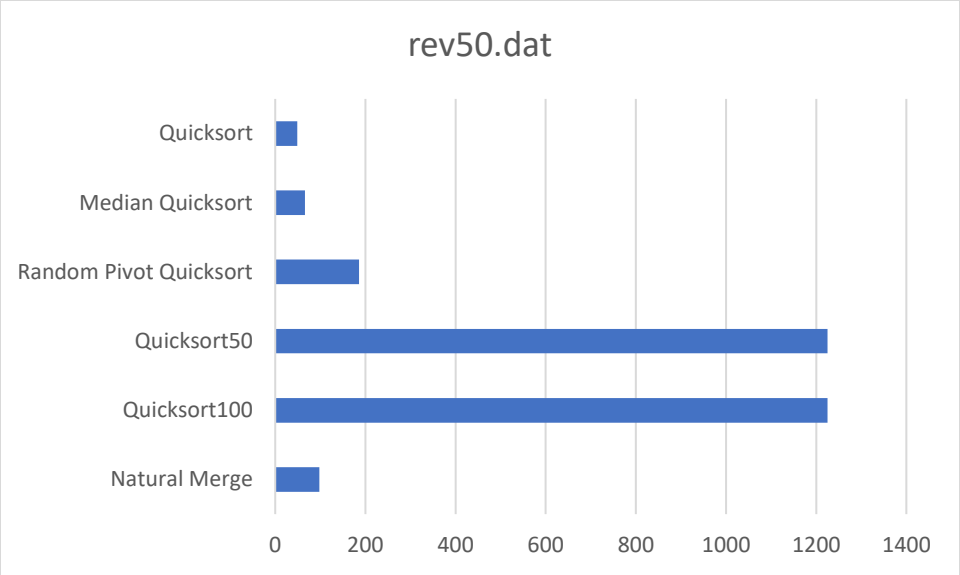

ran50.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
500
1000
1500
2000

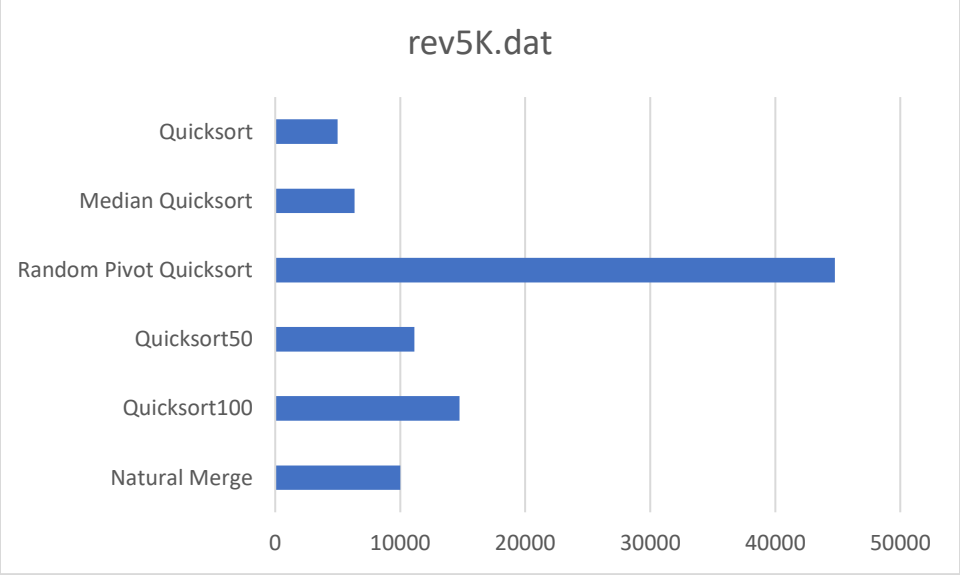

ran5K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
20000
40000
60000
80000
100000


rev10K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
20000
40000
60000
80000
100000
120000


rev1K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
1000
2000
3000
4000
5000
6000
7000
8000


rev2K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
2000
4000
6000
8000
10000
12000
14000
16000


rev50.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
200
400
600
800
1000
1200
1400


rev5K.dat
Quicksort
Median Quicksort
Random Pivot Quicksort
Quicksort50
Quicksort100
Natural Merge
0
10000
20000
30000
40000
50000