Jacob Alspaw

Sorting Algorithms

Descriptions of Algorithms

1. Heap Sort: A sorting algorithm that works by first organizing the data into a special binary tree called a heap. Heaps are binary trees with a priority layout. Some nodes will hold priority over others. In implementation of this method, a lower valued integer held higher priority over a higher valued integer. In our heap, the lowest value always resides in the top node. By relocating this node to our array and reshaping our heap, sorting becomes simple.
2. Quick Sort: An efficient sorting algorithm that will pick a point to pivot across an array. The method will relocate values less than and greater than the pivot to the right and left of the pivot, respectively. This sub-function is call partitioning. Partitioning will continue until the length of the partitioned array is one. At this point, the array is sorted.
3. Merge Sort: A sorting algorithm that utilizes merging subarrays. The algorithm divides the unsorted list into a number of sub-arrays of length one. Repeatedly merging these subarrays, produces new subarrays, until one array remains. Merging combines two arrays into another larger, sorted array.

Results of Report1

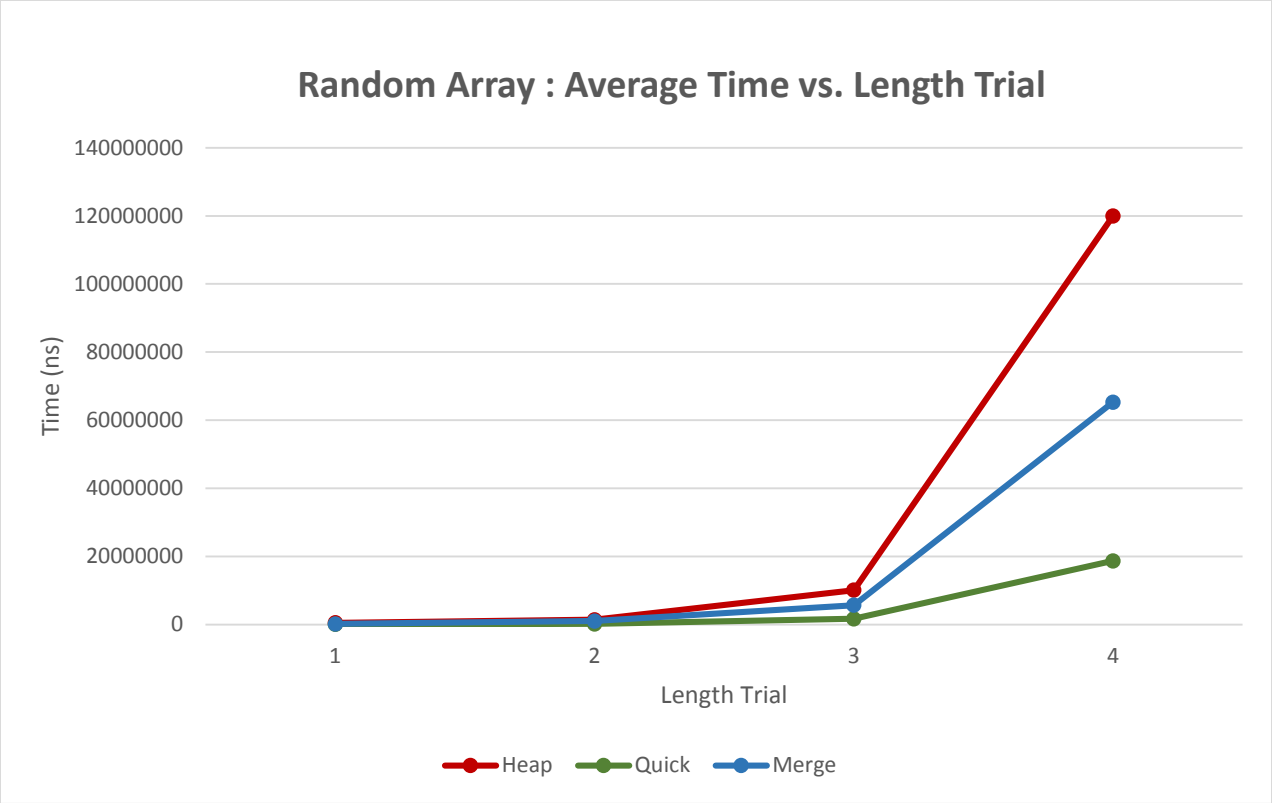
Report1 aimed to generate sorted arrays from three types of initialized arrays. These arrays were randomly generated, presorted, or reverse sorted. Each type of array was tested ten times to eliminate many outlier measurements. Additionally, array size was varied to test groups of length 1000, 10000, 100000, and 1000000. A larger span allowed for observations that could be graphed, comparing run-time and size. Report1 produces an output file labeled "Trial Information.txt", renamed to "Findings.txt" to preserve the data used in this report. This report contains valuable information concerning the functioning of these algorithms on different arrays.

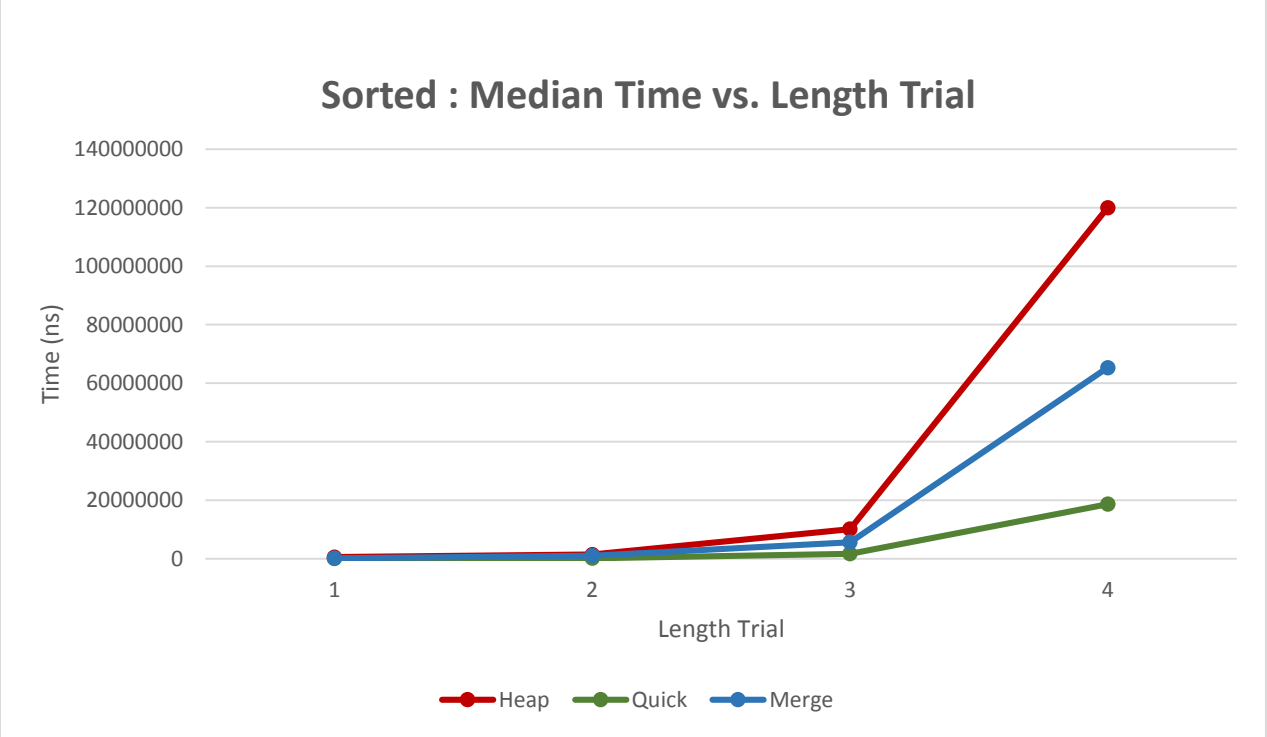
The following table lists results of the mass testing of ten trials for every combination. The total test duration for ten trials is included along with the median time for an individual sorting. The furthest right column ranks their time-efficiency, limiting divisions to the same array type and length (i.e. a test of one method whose length is 1,000,000 and is sorted will not be ranked against another method whose length is 1,000 and is presorted). These results were post-processed by excel.

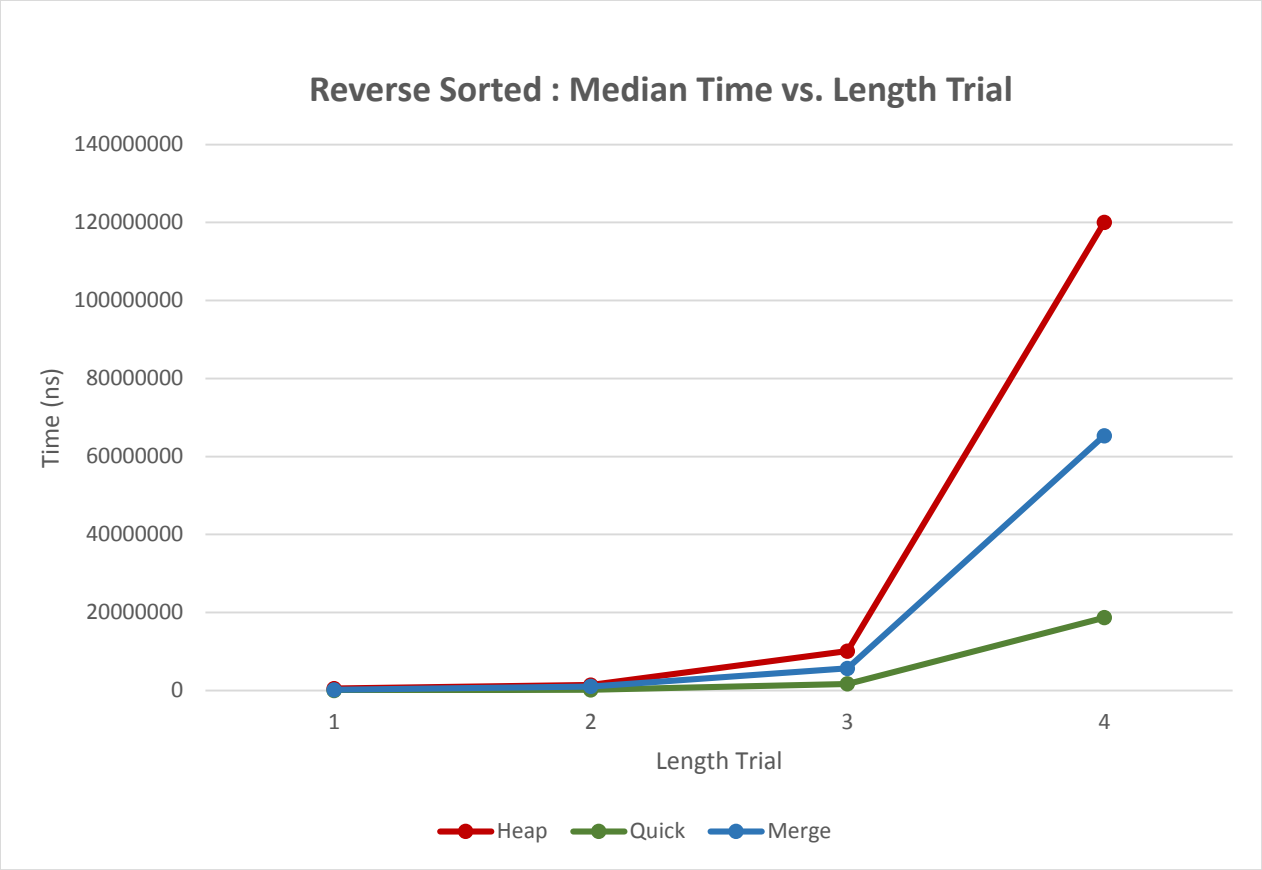
From the table columns labeled "Rank," it should be easy to conclude the efficiency of each algorithm. The quick sort algorithm outperformed its counterparts. In every trial, it handled sorting an array of integers. This can most likely be attributed to a satisfactory solution for finding the pivot point needed to define the location between the two sub arrays. In all our cases, a pivot point value of the average of two points in the array proved efficient in the test cases. This can be confirmed as it won in every ranking category. Merge sort proved to be the second most efficient sorting algorithm. For all but one test, it scored in second place for shortest duration and average execution time. This method's second place ranking can be attributed to its implementation. Merge sort requires merging every sub-length of the array that is a power of two. Quick sort aims to eliminate this unnecessary process by finding a good starting place (i.e. the pivot point). Heap sort was nearly ranked third in every test, but one. Although, this could have been luck on heap sort's part. Heap sort introduced other function calls that increased its run-time. For instance, the need to heapify

and the need to sift introduced additional calculations. Both of which ended up contributing to a longer total duration, longer average sort time, and longer median sort time.

Method	Array Type	Length	Avg. Time(ns)	Median(ns)	Duration(s)	Varience(s^2)	Rank(Avg)	Rank(Med)
Heap	Random	1,000	539007	185978	0.00539007	268.29	2	2
Heap	Random	10,000	1421396	1767406	0.01421396	164.681	3	3
Heap	Random	100,000	10120565	9518538	0.10120565	573.884	3	3
Heap	Random	1,000,000	119999975	109482698	1.19999975	21645.927	3	3
Heap	Sorted	1,000	61582	60351	0.00061582	0.014	3	3
Heap	Sorted	10,000	804631	760744	0.00804631	0.811	3	3
Heap	Sorted	100,000	9785764	8924888	0.09785764	111.348	3	3
Heap	Sorted	1,000,000	153985039	106231574	1.53985039	27257.9	3	3
Heap	Reversed	1,000	83217	58298	0.00083217	0.01	3	3
Heap	Reversed	10,000	1068326	753765	0.01068326	0.264	3	3
Heap	Reversed	100,000	13365448	9878589	0.13365448	4531.226	3	3
Heap	Reversed	1,000,000	151200830	111383943	1.51200830	1016.825	3	3
Quick	Random	1,000	291324	45160	0.00291324	433.757	1	1
Quick	Random	10,000	427502	195010	0.00427502	28.914	1	1
Quick	Random	100,000	2544572	1711571	0.02544572	336.379	1	1
Quick	Random	1,000,000	28203654	18688113	0.28203654	7017.592	1	1
Quick	Sorted	1,000	17078	11496	0.00017078	0.001	1	1
Quick	Sorted	10,000	187579	133428	0.00187579	0.003	1	1
Quick	Sorted	100,000	2249019	1550227	0.02249019	24.514	1	1
Quick	Sorted	1,000,000	27274914	18219678	0.27274914	4733.61	1	1
Quick	Reversed	1,000	17160	12317	0.00017160	0.0	1	1
Quick	Reversed	10,000	201044	139586	0.00201044	0.147	1	1
Quick	Reversed	100,000	2290197	1599492	0.02290197	6.73	1	1
Quick	Reversed	1,000,000	27033512	19922629	0.27033512	1960.069	1	1
Merge	Random	1,000	701461	186389	0.00701461	14.29	3	3
Merge	Random	10,000	1190258	1047306	0.01190258	64.57	2	2
Merge	Random	100,000	7574022	5670479	0.07574022	195.85	2	2
Merge	Random	1,000,000	106446659	65316842	1.06446659	443.552	2	2
Merge	Sorted	1,000	43887	34897	0.00043887	0.001	2	2
Merge	Sorted	10,000	570824	440107	0.00570824	0.185	2	2
Merge	Sorted	100,000	7393628	5190139	0.07393628	38.738	2	2
Merge	Sorted	1,000,000	95641178	61598105	0.95641178	115.998	2	2
Merge	Reversed	1,000	46884	34897	0.00046884	0.001	2	2
Merge	Reversed	10,000	614301	460224	0.00614301	0.06	2	2
Merge	Reversed	100,000	7123118	5278406	0.07123118	1.416	2	2
Merge	Reversed	1,000,000	94322993	65489682	0.94322993	6.095	2	2


Random Array : Average Time vs. Length Trial
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
Time (ns)
1
2
3
4
Length Trial
Heap
Quick
Merge


Sorted : Median Time vs. Length Trial
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
Time (ns)
1
2
3
4
Length Trial
Heap
Quick
Merge



Random Array: From the graph of random arrays' sort times and the length, we can see all algorithms have nonlinear sorting run-time. In this visual representation, merge sort seems to be pressed between quick sort and heap sort.

Presorted Array: From the graph of sorted arrays' sort times and the length, we can see all algorithms have nonlinear sorting run-time. In this visual representation, merge sort seems to be pressed between quick sort and heap sort.

Reverse Sorted Array: From the graph of reverse sorted arrays' sort times and the length, we can see all algorithms have nonlinear sorting run-time. In this visual representation, merge sort seems to be pressed between quick sort and heap sort.

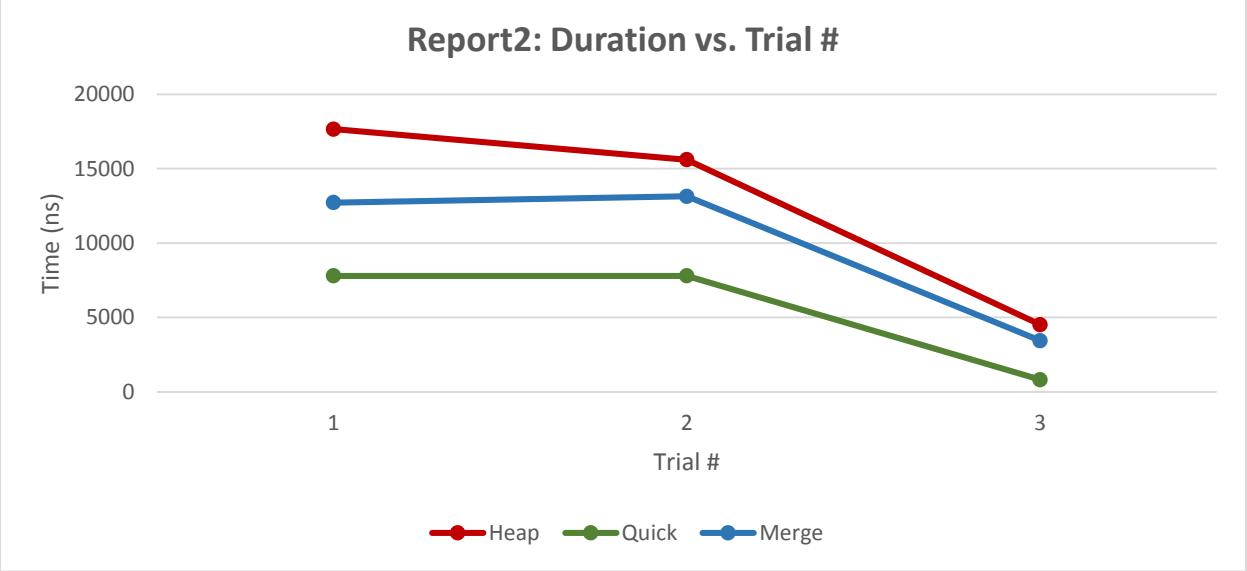
Overall, merge sort seemed to produce the least variance, especially when approaching larger numbers. Of the 12 trials, merge sort produced the smallest variance in about 75% of cases. In second came quick sort, and in a distant third was heap sort. These rankings were given in order of least variance to largest variance. It makes sense that merge sort was the most predictable time sequence because it does the same sequence again and again independent on array type.

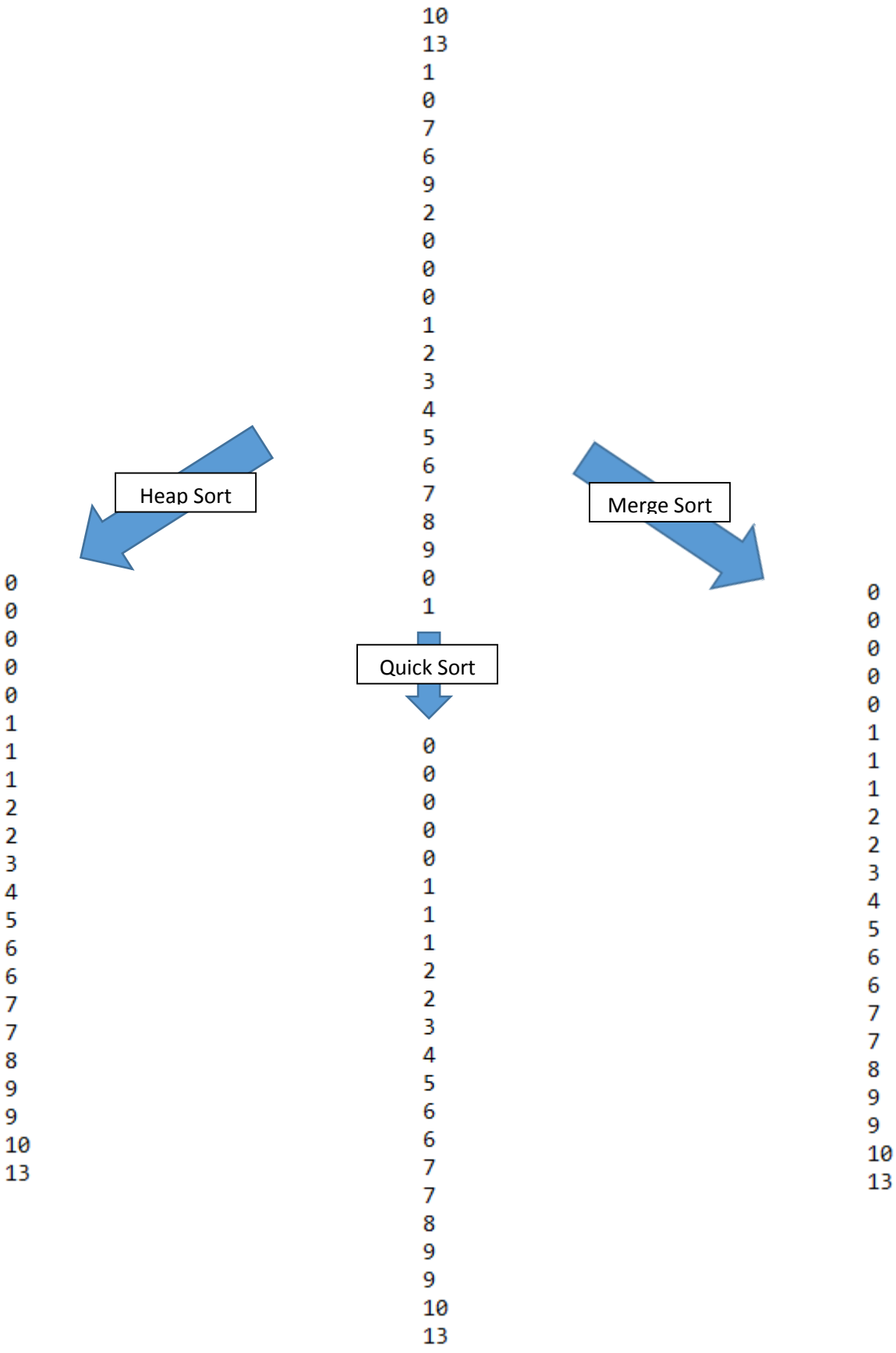
Results of Report2

Report2 aimed to read a file and transfer its contents into an array. This array would be sorted three times by each method for a total of nine sorts. The array will be written back into files whose names include the maker's CWRU ID (jaa134) followed by an abbreviation of the method (HS, QS, MS). My instantiation of Report2 creates files, containing sorted arrays, named "jaa134HS.txt", "jaa134QS.txt", and "jaa134MS.txt". Of the three runs, the median recorded run-time will be reported to the system in an arbitrary form; the form consists of the same abbreviation of the method used immediately followed by the maker's CWRU ID.

The following are an actual result printed to the System:
HSjaa134 = 17653ns; QSjaa134 = 7800ns; MSjaa134 = 12727ns
HSjaa134 = 15600ns; QSjaa134 = 7801ns; MSjaa134 = 13138ns
HSjaa134 = 4516ns; QSjaa134 = 821ns; MSjaa134 = 3443ns

Method and Trial #	Time(ns)	Time(s)
Heap Sort #1	17653	0.000017653
Quick Sort #1	7800	0.000007800
Merge Sort #1	12727	0.000012727
Heap Sort #2	15600	0.000015600
Quick Sort #2	7801	0.000007801
Merge Sort #2	13138	0.000013138
Heap Sort #3	4516	0.000004516
Quick Sort #3	821	0.000000821
Merge Sort #3	3443	0.000003443




10
13
1
0
7
6
9
2
0
0
0
1
2
3
4
5
6
7
8
9
0
1
Heap Sort
Merge Sort
Quick Sort
0
0
0
0
0
1
1
1
2
2
3
4
5
6
6
7
7
8
9
9
10
13
0
0
0
0
0
1
1
1
2
2
3
4
5
6
6
7
7
8
9
9
10
13
0
0
0
0
0
1
1
1
2
2
3
4
5
6
6
7
7
8
9
9
10
13

Reporting2 shares the same results as Reporting1. All arrays were sorted in a timely manner and displayed in an output file containing my CWRU ID and an abbreviation for the sorting algorithm. The report then correctly printed to the system the median of three trials. Through three consecutive test runs of report two (included in the table and graph above), our values match that of Report1's findings. Quick sort out-performed merge sort and heap sort algorithms and merge sort was seemingly the close tothe speed of quick sort and heap sort.