# Runtime and Operation Analysis of Sorting Algorithms

By Quinn Bendelsmith

Over the course of the semester, we have spent a lot of time exploring sorting algorithms and their efficiency in terms of both space, operations, and runtime. As a result of this, we have spent the last couple weeks exploring the theories we have discussed in class and put them to the test by running the sorting algorithms with Java code to analyze operations and runtime of the sorting algorithms we have discussed in class. We have omitted bubble sort and selection sort due to their overall inefficiency in both runtime and management of space. That left us with insertion sort, quick sort, merge sort, counting sort, and lastly radix sort. We were asked to push our computers to their theoretical limits to analyze a wide range of array and integer sizes. Unfortunately for me, my computer does not contain a lot of space and runs slowly, so my limit was rather small compared to others and higher functioning computers. My laptop capped out at array sizes of just 100,000 integers. It was, however, capable of working with integers up to 199,999,999. During the testing phase of my computers limit, I tested from 1,000,000 to 750,000, to 500,000 and it finally worked with 100,000 but took a long time to run unless I closed out all applications on my computer, which I was not always able to do, due to this computer being used for work. This was after expanding heap and stack to 4 and 1 gigabytes respectively. Now having discussed the limits of my research, it is time to look at the findings.

First, we will examine insertion sort. I ran 3 experiments with insertion sort, first, insertion sort on an already sorted list, second, insertion sort on a list in reverse order and last, the average of three random lists. For insertion sort, I thought it would be more helpful to look at the operations than the runtime, because there were some inconsistencies with the first run of sorting algorithms. This led to the time when sorting at 10 elements to take longer than that of

500 elements, which was counterintuitive and was most likely a result of how the code compiled. We know from previous class discussions that an already sorted list is the best-case scenario for insertion sort and we would expect it to run at about Omega(n). As is evident from Figure 1, the operations for an already sorted list was f(n) = n-1 for array sizes ranging from 10 to 100,000. Despite the appearance of the graph, it is in fact a linear graph following the equation n-1 as you can see from the labels. In Figure 2, we look at insertion sort on a reverse ordered list, which we have determined in class to be the worst-case scenario and the average of 3 randomly ordered lists. For the worst-case scenario we expect the operations to be approximately f(n) = ½($n^2$-n). And with a random list, I calculated the comparisons should be approximately f(n) = ¼($n^2$-n). As can be seen in both Figure 2 and Table 1, these expected equations for the number of operations are fairly close to our results, supporting our claims in class that worst-case and average case for insertion sort, when n gets larger, is dominated by the $n^2$, thus we would expect the runtime to be both Theta($n^2$) and O($n^2$).

Second, let's look at quick sort. We tested three different forms of arrays, which were, already sorted simples, already sorted integers, and randomly sorted large integers, which means between 100,000,000 and 199,999,999. Much like insertion sort, we will look at the operations again due to some inconsistencies in runtime. This seemed to be a common occurrence with my laptop, so looking at operations is just significantly more reliable than runtime. From our discussions in class, no matter how the list is ordered, with an efficient method of picking the pivot value, the runtime and operations should approximately be O(nlgn). As we can tell from Figure 2, there is not a huge variation between the graphs and that is to be expected, because the use of getting 3 numbers and finding the middle number is a very good way to find a good pivot. Looking at Table 2, I added the expected number of operations, which was determined to f(n) =

nlgn. While for all cases, the number of operations was greater than the expected average, that can be expected from only running the experiment once. If we were to run it multiple times and average the results, it would be closer to the expected number of operations. Due to the randomness of picking a pivot value, there should be variance across the board while running this experiment, but it does follow the general form of the expected operations formula of f(n) = nlgn and most likely there is a constant before n.

Third, we will look at merge sort on already sorted lists, incorporating insertion sort once the subarrays are less than or equal to 1000, 100, and lastly, a full merge sort where we do not use insertion sort. As discussed in class, merge sort will run approximately at O(nlgn) and the operations equation follows the general function of f(n) = 2nlgn for a full insertion sort. Since the lists are already sorted, we would expect insertion sort to run at f(n) = n-1 as was determined both in class and earlier in insertion sort. As is evident from Figure 4, the sooner we shift to insertion sort, the more efficient and fewer operations it will take. This is the case be insertion sort running at Omega(n) will require fewer operations than merge sort running at O(nlgn). While we did not look at non-sorted lists, using insertion sort at a certain threshold is more cost-effective than using merge sort the whole way. This is since insertion sort operates on the same array, while merge sort operates on many arrays of smaller sizes at each iteration, and then it merges back to the original array but now sorted. This would be an interesting way to expand our experiment in this section.

Last, we look at counting sort and radix sort with large integers ranging from 100,000,000 and 199,999,999. For counting sort, we have determined the runtime and operations to be O(n+k) where n is the number of elements and k is the range of non-negative values. Since my arrays capped out at 100,000 elements, the operations will be dominated by the k value of

99,999,999. The equation follows f(n) = (3n – 1) + 4k for counting sort in this case. But the operations will be dominated by the 4k, thus counting sort will run approximately at O(k). For radix sort with have determined that the runtime will be O(n*k) where n is the number of elements in the array and k is the number of digits in the max value. In this case, that would be 9. The equation I found for the number of operations for radix sort was f(n) = 2*n*k + 114. This does follow the general form of O(n*k). Unfortunately, as discussed with you in class, my radix sort fails to sort, so I am not 100% certain that the operations are correct, but they follow the structure of what we would expect of radix sort's runtime.

In summation, we have justified the amount of time we spent on sorting algorithms throughout this class by putting them to the test and have our analysis confirm the theoretical functions that we discussed in class. It was interesting to examine both the operations and runtimes of different sorting algorithms. Seeing how efficient algorithms are when dealing with randomly sorted, reverse sorted, and sorted arrays was helpful for broadening my understanding of sorting algorithms by seeing them in action. If I were to run this experiment again, I would make sure to do it on a powerful desktop computer to allow for a larger sample size, as my laptop let me down in terms of capability.

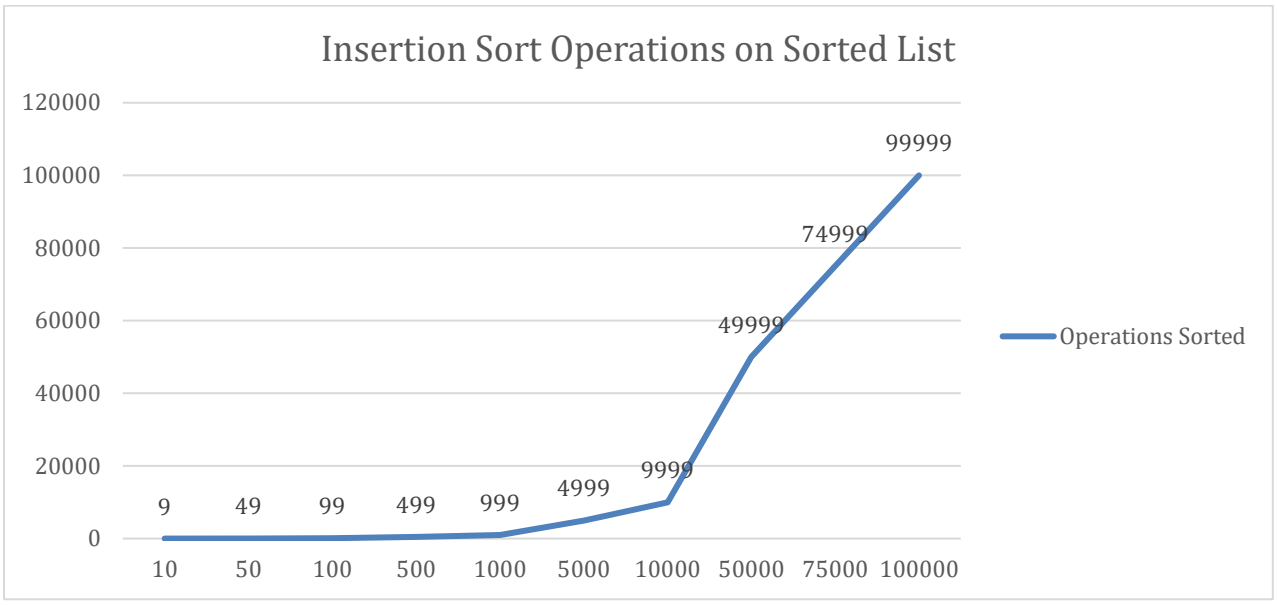


**FIGURE 1**

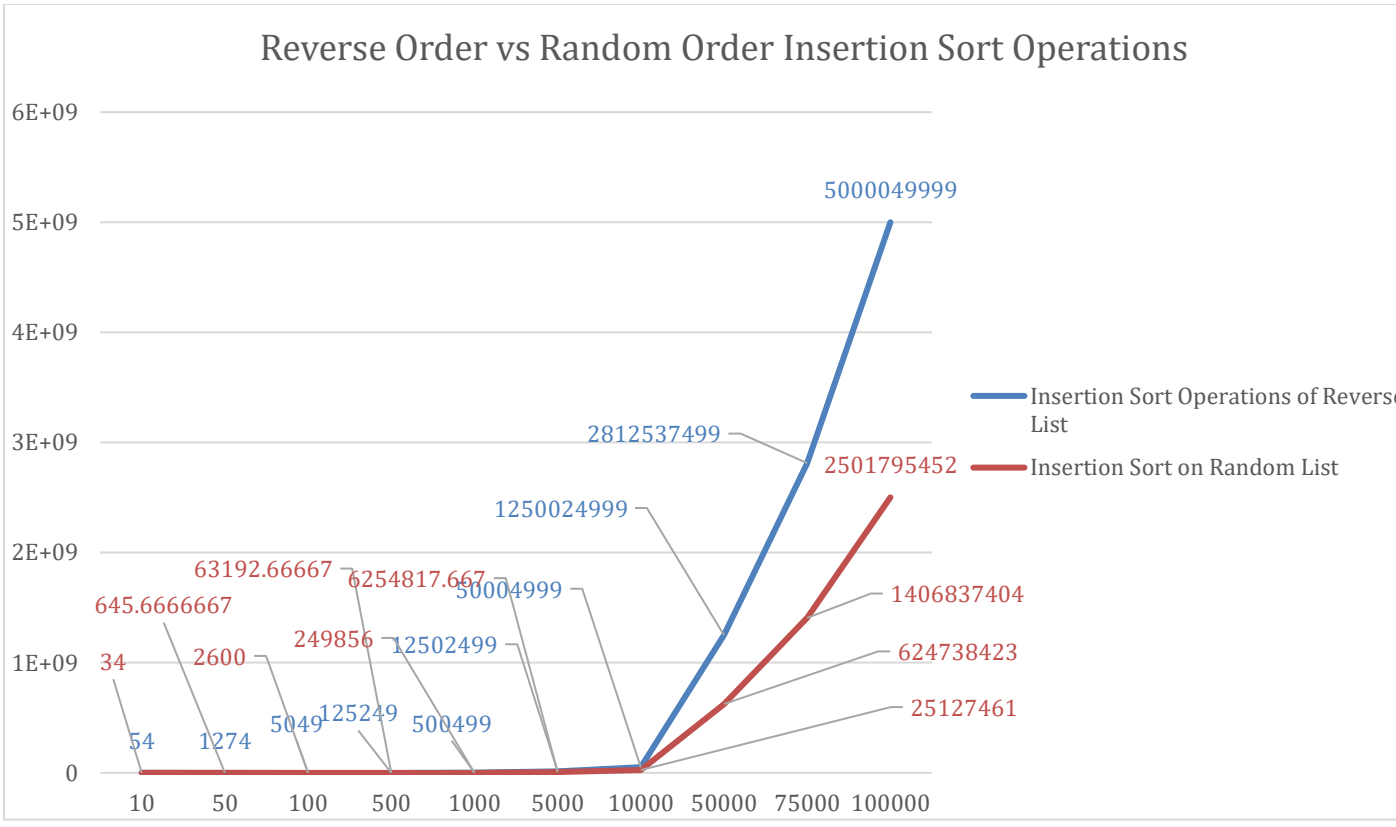


**FIGURE 2**

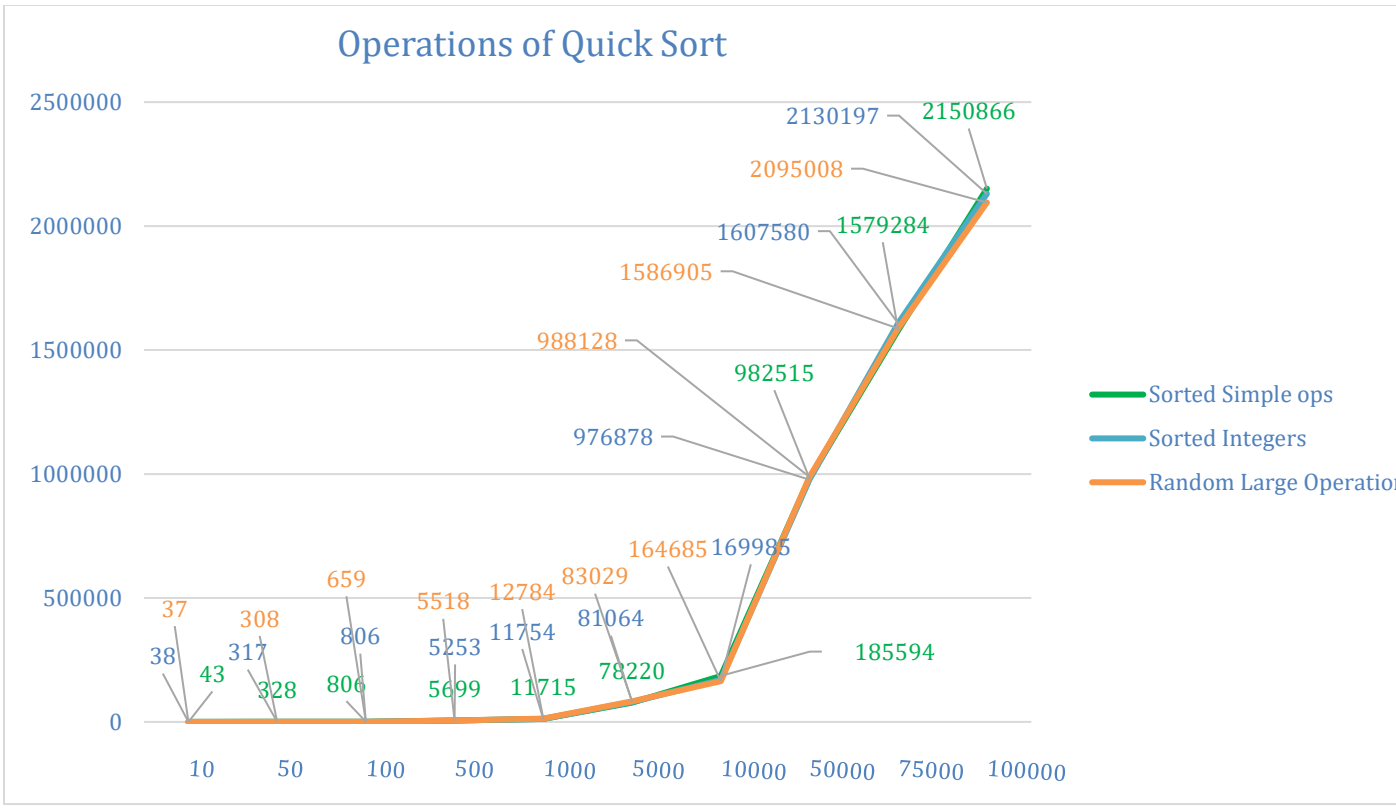
Operations of Quick Sort
2500000
2000000
1500000
1000000
500000
0
10
50
100
500
1000
5000
10000
50000
75000
100000
2130197
2150866
2095008
1607580
1579284
1586905
988128
982515
976878
164685
169985
659
5518
12784
83029
37
308
806
5253
11754
81064
38
317
43
328
806
5699
11715
78220
185594
Sorted Simple ops
Sorted Integers
Random Large Operatio

**FIGURE 3**

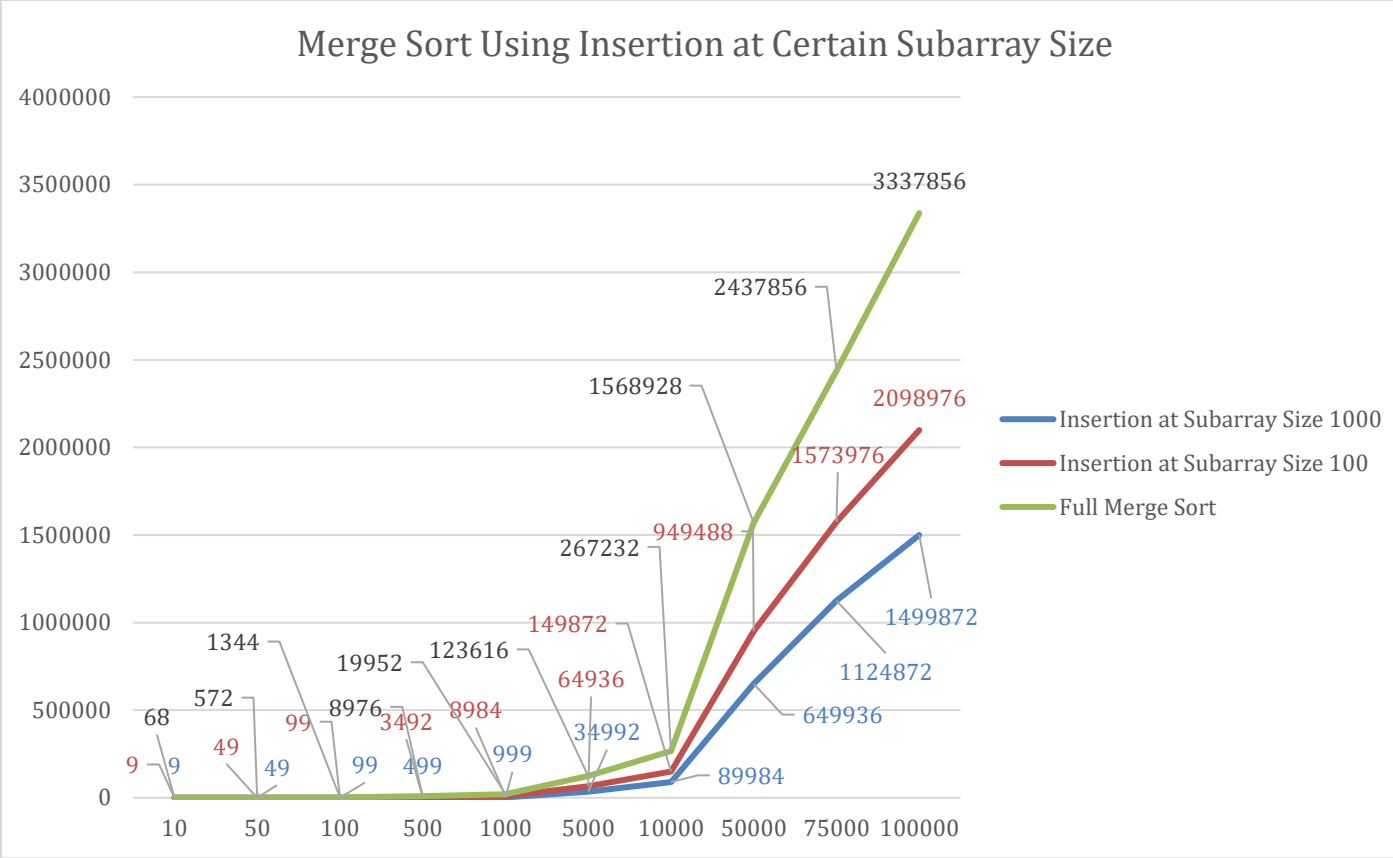


**FIGURE 4**

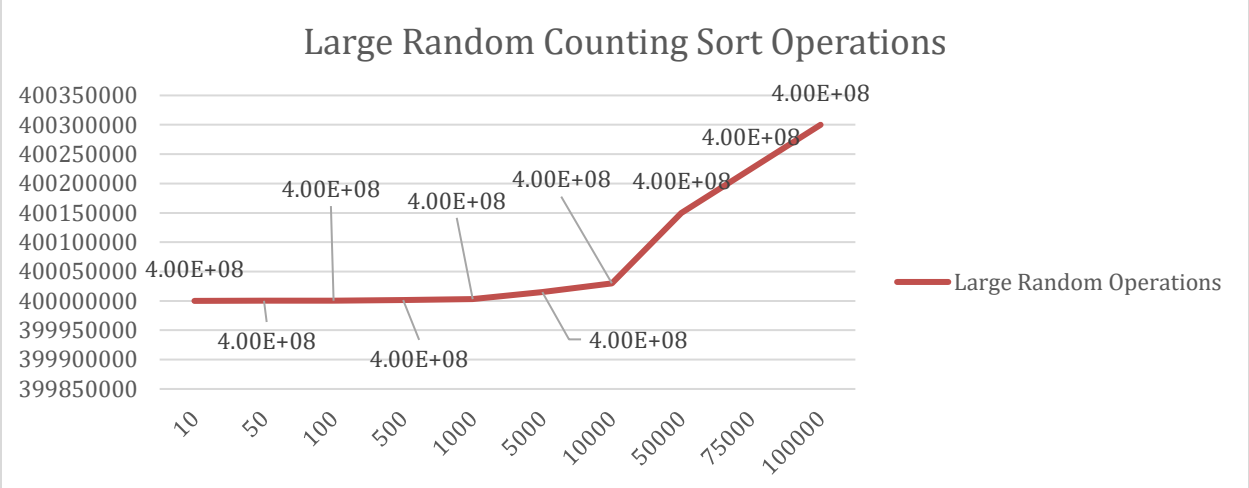


**FIGURE 5**

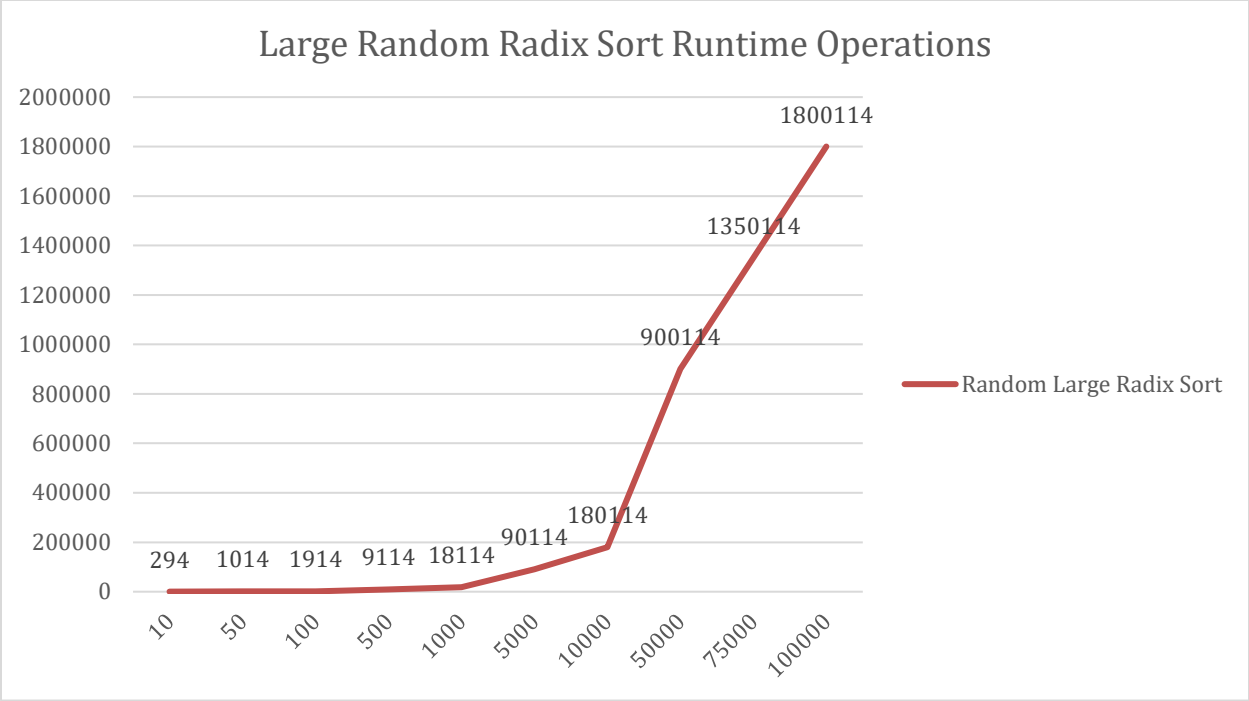


**FIGURE 6**

| Array Size | Ops Reverse | Ops Reverse Expected | Ops Random | Ops Random Expected |
|---|---|---|---|---|
| 10 | 54 | 45 | 34 | 22.5 |
| 50 | 1274 | 1225 | 645.6666667 | 612.5 |
| 100 | 5049 | 4950 | 2600 | 2475 |
| 500 | 125249 | 124750 | 63192.66667 | 62375 |
| 1000 | 500499 | 499500 | 249856 | 249750 |
| 5000 | 12502499 | 12497500 | 6254817.667 | 6248750 |
| 10000 | 50004999 | 49995000 | 25127461 | 24997500 |
| 50000 | 1250024999 | 1249975000 | 624738423 | 624987500 |
| 75000 | 2812537499 | 2812462500 | 1406837404 | 1406231250 |
| 100000 | 5000049999 | 4999950000 | 2501795452 | 2499975000 |

**TABLE 1**

| Size | Expected Operation | Simple Sorted | Integer Sorted | Random Large Sorted |
|---|---|---|---|---|
| 10 | 33.21928095 | 43 | 38 | 37 |
| 50 | 282.1928095 | 328 | 317 | 308 |
| 100 | 664.385619 | 806 | 806 | 659 |
| 500 | 4482.892142 | 5699 | 5253 | 5518 |
| 1000 | 9965.784285 | 11715 | 11754 | 12784 |

| | | | | |
|---|---|---|---|---|
| 5000 | 61438.5619 | 78220 | 81064 | 83029 |
| 10000 | 132877.1238 | 185594 | 169985 | 164685 |
| 50000 | 780482.0237 | 982515 | 976878 | 988128 |
| 75000 | 1214595.223 | 1579284 | 1607580 | 1586905 |
| 100000 | 1660964.047 | 2150866 | 2130197 | 2095008 |

**Table 2**