Lab 12 Report:

By: Sam Graler and Randy Hucker (Group 25)

Instructions for Running / Compiling:
.cpp files for all Tasks are located in the submitted visual studio project. Open / run the visual studio project as normal.

Work was divided evenly between each group member (50% for Sam and 50% for Randy)

Write a Lab Report that includes the following information:
Use the output of this to make a table similar to (these numbers are completely fabricated) the following with the values showing the average of all your runs for this test type:

Time in Nanoseconds:

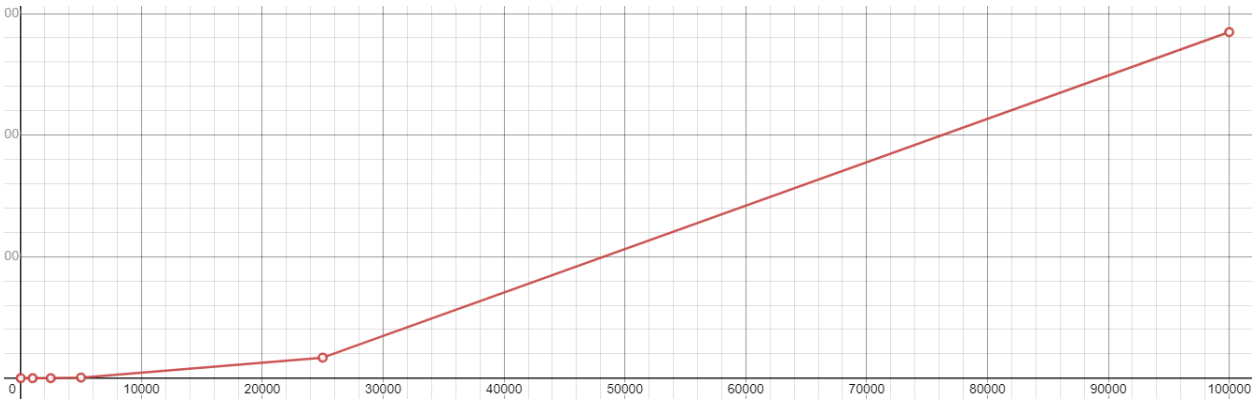
```
Time in Nanoseconds
Sizes:          10      100     500     5000            25000           100000

Bubble Sort:    470     18990   481900  47203230        1688135110      28476376230
Insert Sort:    1070    9120    294500  19019490        420619920       6426919270
Quick Sort:     640     7450    56700   815740          4531490         19051270
Merge Sort:     5680    53160   334240  3628460         18913560        70712220
Count Sort:     640     2890    12200   107540          484240          1886010
Radix Sort:     1670    9570    44930   532930          3732640         15326600
Heap Sort:      430     8980    99550   1172830         7096140         31052000
```

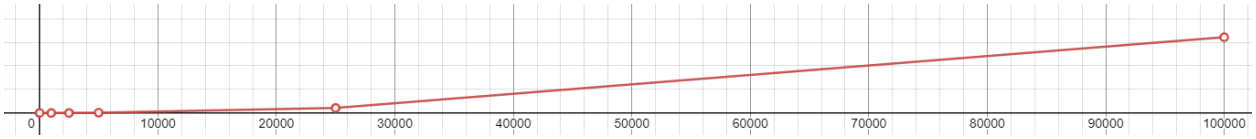
Time in MilliSeconds:

	10	100	500	5000	25000	100000
Bubble Sort	0.00047	0.01899	0.4819	47.20323	1688.13511	28476.37623
Insertion Sort	0.00107	0.00912	0.2945	19.01949	420.61992	6426.91927
Merge Sort	0.00568	0.05316	0.33424	3.62846	18.91356	70.71222
Quick-Sort	0.00064	0.00745	0.0567	0.81574	4.53149	19.05127
Heap Sort	0.00043	0.00898	0.09955	1.17283	7.09614	31.052
Count Sort	0.00064	0.00289	0.0122	0.10754	0.48424	1.88601
Radix Sort	0.00167	0.00957	0.04493	0.53293	3.73264	15.3266

a. Explain how well or poorly it matches your expectations for performance given the known Big O notation for the given sort algorithms. Include what you expected for time for each of the array sizes based on the results for array size of 10.
    i. Bubble Sort:
        1. Average/Worst Case Performance - O(N^2)



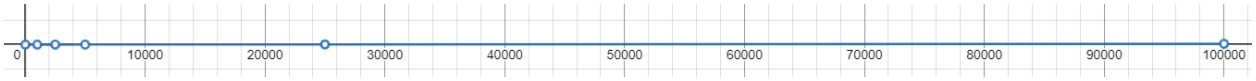
        2. From the graph above, we can see that as we add more elements to the sorting algorithm, it does look like the parabolic curve that is represented by N^2. The results of our algorithm performed as expected, the more elements we added, the slower the cycles. At array size 10, the bubble sort was extremely fast, but this quickly deteriorated.
    ii. Insertion Sort:
        1. Average case - $O(n^2)$. Typically performs 2-3 times faster than bubble sort



        2. As we can see from this result's graph, it also resembles a N^2 parabolic curve. In class we discussed that this algorithm should perform about 2/3 times faster than the bubble sort, but still have the same curvature… this happened to be the case in our results. We can very obviously see the curve beginning to form, but at a scale about 2-3 times smaller.

iii. Merge-Sort:

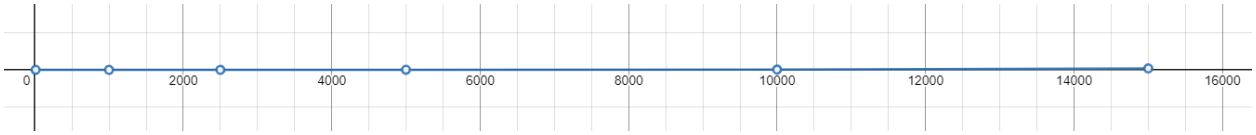
1. Performance: O(nLog(n))



2. Our merge sort seemed to operate much faster than what we discussed in class. Our discussion led to the conclusion that we should see nlogn performance, however our graph shows O(logn) performance. We are a little confused as to why this might be the case. Merge sort should have nLogn performance because of the memory that is required for recursive calls, but overall, we were both happy and surprised with our merge sort's performance. From doing a comparison calculation, we can tell that our results are close to the nlogn number which should be around 20 times slower from 10 to 100 elements, in our case our 10 element number was .00568ns. Multiplying that by 20 yields 0.1136ns, which is much larger than our 0.05316ns result. Overall, we seem to be following the nlogn curve, but overall the sort was faster than expected.

iv. Quick Sort

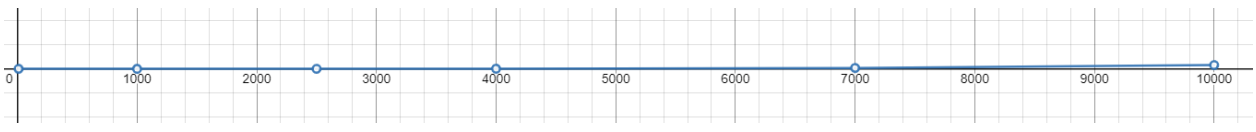
1. Avg/Best Case - O(nlogn)



2. We reduced the scale for the quick sort numbers due to how close they were to the axis line, but we seemed to have the same results as the merge sort performance. We were expecting more of a parabolic curve, as that is what nlogn looks like, but we are getting these more linear results. We expect that if we were to add an arbitrary amount of extra elements, we could begin to see that parabolic curve, but using the results from the lab, we look to almost be having O(1) results - where the results form an almost horizontal line. From doing a comparison calculation, we can tell that our results are close to the nlogn number which should be around 20 times slower from 10 to 100 elements, in our case our 10 element number was .00064ns. Multiplying that by 20 yields 0.0128ns, which is much larger than our .00745ns result. Overall, we seem to be following the nlogn curve, but at pace slower than expected.

v. Heap-Sort

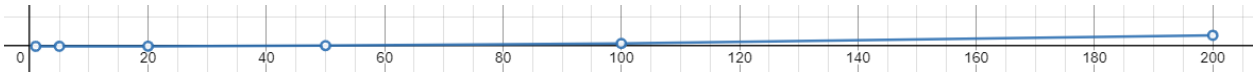
1. Average/Best/Worst - O(nlogn)



2. We reduced the scale for the heap sort numbers due to how close they were to the axis line, however, we can see the nlogn line forming. We see the line diverge from the x-axis near the value for our 100000 results, so we are finding the curve that is represented by nlogn, but just at a much slower rate than expected. We imagined the curve would be much more prominent, but it is instead quite small.

vi. Counting Sort

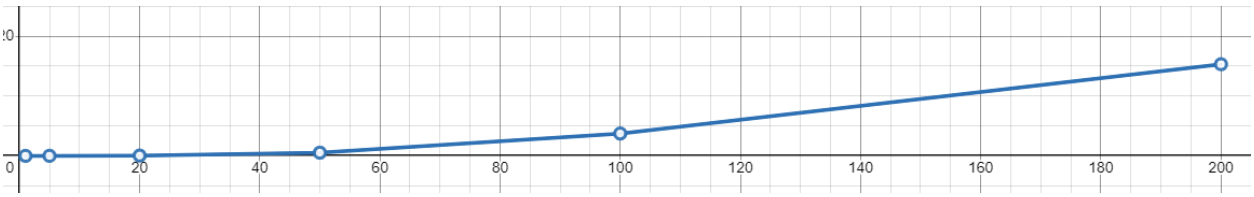
1. Best/Worst/Average - O(2n+k) where n is the item to sort and k is the range of key values.



2. Our results for counting sort were the fastest of any of the tests. This was surprising to us because the count sort performs best when multiple of the same value are used to sort, however, since we are using random numbers with a range of 2n in our arrays, duplicates are not extremely likely. We do start to see the O(2n+k) in our graph, not as prominent as a true 2n graph would look like, but we do see the more linear results in this graph. It's quite close to the graph we expected.

vii. Radix-Sort

1. Best/Worst/Average - O(d*n) where d is the number of digits if d is small(ish) this is O(n)



2. Our results for radix sort, which were enlarged here for visibility, show the linear result quite well. Since the number of digits was randomized, we weren't sure what to expect from this result, but we were quite pleased when we found that our graph pretty accurately followed the O(n)/O(dn) trend line. We expected this to follow the results from the counting sort closely, which it did. Our 10 elements (0.00167ms) sorted about 10 times faster than 100 (0.00957ms), and our 100 elements sorted about 5 times faster than 500(0.04493ms), and so on.