Team Information and Sources:

Team Members: Casey Quinn, Matt Juntunen
Sources: Merge Sort Algorithm in Java - Full Tutorial with Source ,
Counting Sort Sorting Algorithm (Working with Diagram) | Part - 1 | Sorting Algorithms - DSA
, https://www.geeksforgeeks.org/chrono-in-c/, MergeSort in C++ ,
https://en.wikipedia.org/wiki/Merge_sort,
https://www.baeldung.com/cs/bubble-sort-time-complexity,
https://mathbits.com/MathBits/Java/arrays/Exchange.htm, https://en.wikipedia.org/wiki/Quicksort
https://www.programiz.com/dsa/quick-sort

Introduction:

In this assignment, we were tasked with selecting six sorting algorithms to implement in the C++ programming language and perform experimental analysis with unsorted inputs of integers of varying sizes. We had to select two algorithms from the following three categories: simple sorting algorithms, non-comparative sorting algorithms, and divide-and-conquer sorting algorithms.

We chose to implement the following algorithm: bubble sort, exchange sort, counting sort, radix sort, merge sort, and quick sort. To structure the code, we used the given templated SortLib library containing the function declarations. We then wrote the source code for each of the six algorithms we chose in C++ source files. In order to perform the experimental analysis, we wrote the sort.cpp source file which was capable of reading input from a user or a file and outputting to the console or an output file. We used the chrono library in C++ to do the time analysis.

Bubble sort is a simple comparative sorting algorithm that works by continuously comparing adjacent elements in an array and swapping them if they are in the wrong order in the list. The process continues until the list is sorted. Exchange sort is very similar to bubble sort. It also compares elements of an array and performs swaps if they are out of order; however, exchange sort compares the first element with each subsequent element in the array and makes the necessary swaps. After the first pass, the exchange sort then takes the second element and compares it to every other non sorted element.

Counting sort is a non comparative sorting algorithm in which you count the number of times each key occurs and place the count in a separate array at the appropriate index. After this you compute a prefix sum which allows you to loop over the original unsorted array backwards and use the new array with the positions to output to a sorted array. Counting sort is frequently used as a subroutine in the radix sorting algorithm. For the radix technique, you only create an additional array containing the digits 0-9 and then sort the numbers in the array based on each place value from right to left. The main difference between the two is that radix uses less additional space but has the trade-off of typically having a slower runtime.

Both merge sort and quicksort are classified as being divide and conquer algorithms. The basic idea with these types of solutions is to continuously divide a problem into two or more sub-problems of the same or related type until they become simple enough to solve directly. For these methods we typically use recursion to implement the algorithm. For the merge sort we continuously divided the inputted array into subarrays until there was only one element in the array. Then we used a merge function to merge the two halves of the original array so that is sorted. This is known as the top-down implementation. Quicksort works by selecting a pivot element from the array and partitioning the other elements into two additional arrays. One of the subarrays contains elements that are less than the partition element and the other contains the elements that are greater. The subarrays are then sorted recursively.

Theoretical Analysis:

Complexity	Best	Average	Worse
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Exchange Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$
Radix Sort	$O(d(n + k))$	$O(d(n + k))$	$O(d(n + k))$
Merge Sort	$O(n\, log\, n)$	$O(n\, log\, n)$	$O(n\, log\, n)$
Quicksort	$O(n\, log\, n)$	$O(n\, log\, n)$	$O(n^2)$

Bubble Sort: The number of comparisons is found with the two loops. Each loop has a time complexity of $T(n)$. When nested, the time complexity becomes $T(n^2)$. This complexity is the same for best, average, and worst cases, making the it $O(n^2)$ for best, worst and average cases.

Exchange Sort: The number of comparisons is found with the two loops. Each loop has a time complexity of $T(n)$. When nested, the time complexity becomes $T(n^2)$. This complexity is the same for best, average, and worst cases, making it $O(n^2)$ for best, worst and average cases.

Counting Sort: There are a total of 5 loops, all non nested, in this program. Four of them iterate $T(n)$ times. The other loop is up to max, so this program will run $T(n + k)$ times, making the big O is $O(n + k)$ for best, worst and average cases..

Radix Sort: Since radix sort uses counting sort, the time complexity of the counting sort is $T(n + k)$ as stated in the previous section. Let d be the number of cycles that the algorithm runs. Therefore the big O is $O(d(n + k))$ for best, worst and average cases.

Merge Sort: Because merge sort halves each iteration and continues to recursively do this to the whole array, the complexity is $T(logn)$ as it halves. It “backs out” of the recursion n times to reach the fully completed list, making the big O = $nlogn$ for best, worst and average cases.

Quicksort: The worst case for this algorithm is $T(n^2)$, if the pivot element is an extreme of the array, leaving merge sort to split the remaining array into an empty list and a list of n-1 elements. Therefore the worst case is big O = n^2. When the selected pivot is a middle number in the range of numbers, the program will call the recursion on equal lengths of sub arrays, resulting in $T(nlogn)$. Therefore the big O = $(nlogn)$ for best and average cases.

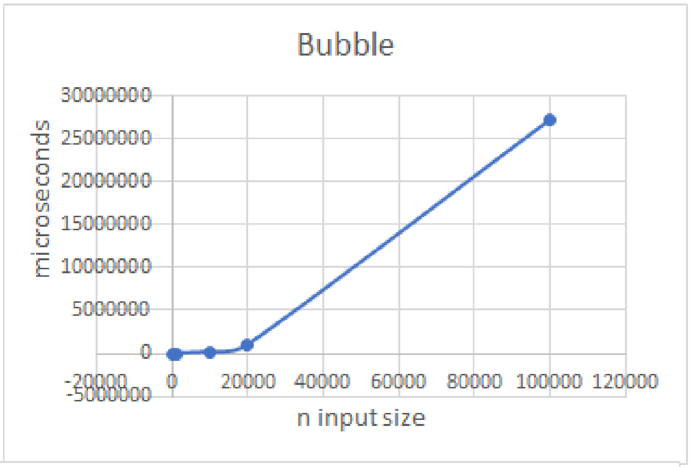
Experimental Results:

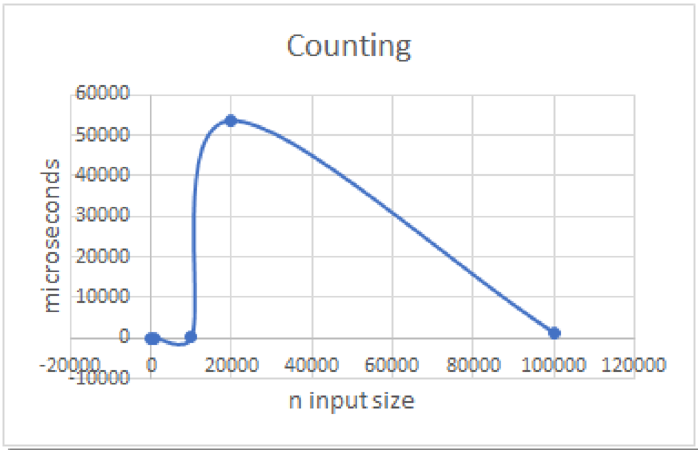
Machine Specifications:

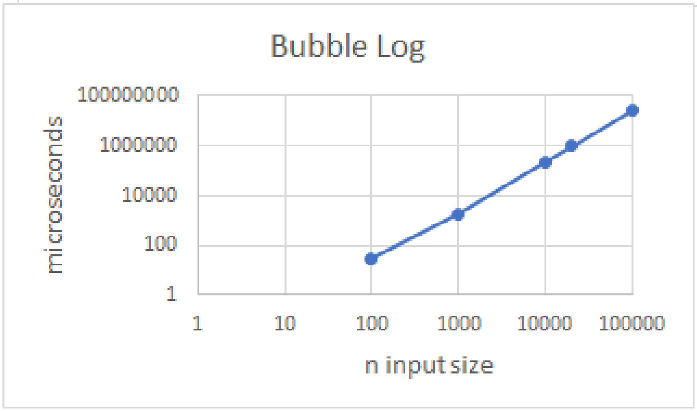
- 2017 MacBook Pro
- Processor: 2.3 GHz Dual-Core Intel Core i5
- Memory: 8 Gb 2133 MHz LPDDR3
- Graphics: Intel Iris Plus Graphics 640 1536 MB

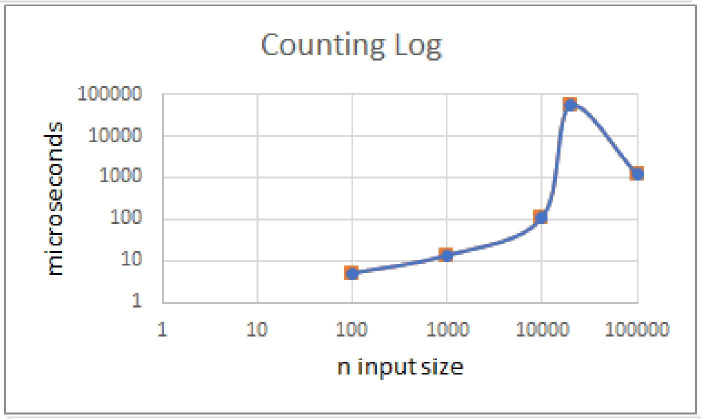
Results Table (in microseconds)

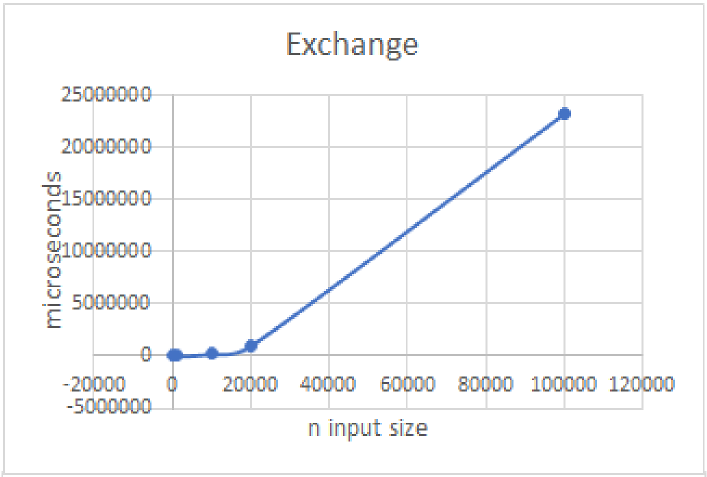
Algorithm	10^2			10^3			10^4			2×10^4			10^5		
	Trial 1	Trial 2	Trial 3	Trial 1	Trial 2	Trial 3	Trial 1	Trial 2	Trial 3	Trial 1	Trial 2	Trial 3	Trial 1	Trial 2	Trial 3
Bubble	30	30	31	1786	1949	1945	226904	237252	226482	968736	1.01e6	1.02e6	2.84e7	2.66e7	2.67e7
Exchange	33	32	33	3233	3497	3579	248574	248299	248929	964560	965230	968088	2.51e7	2.21e7	2.22e7
Counting	5	6	4	13	12	15	128	94	96	49730	59583	51960	1557	1073	1060
Radix	16	16	16	272	293	271	2606	2497	2425	12813	13030	12901	32882	33224	33196
Quick	9	7	7	91	113	111	1670	1335	1365	2193	2201	2364	12774	12859	12858
Merge	38	57	38	382	427	349	3444	3293	2781	5615	7190	6690	31210	32143	31822

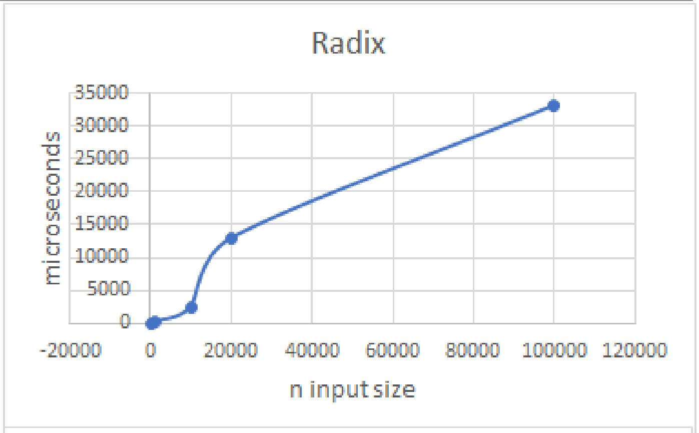

Bubble
microseconds
30000000
25000000
20000000
15000000
10000000
5000000
0
-5000000
-20000
0
20000
40000
60000
80000
100000
120000
n input size

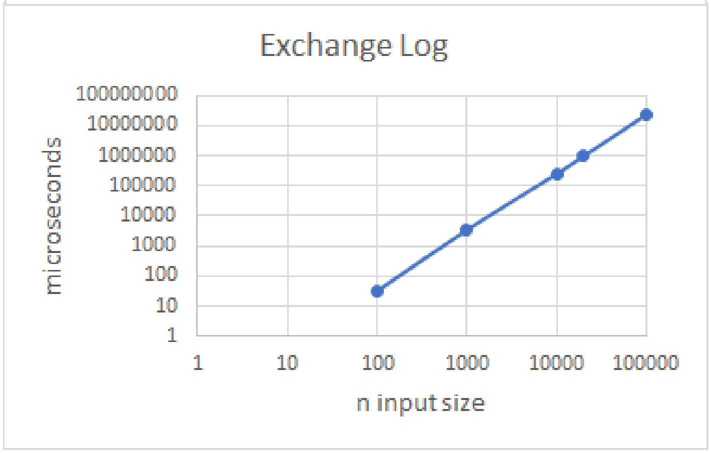

Counting
microseconds
60000
50000
40000
30000
20000
10000
0
-10000
-20000
0
20000
40000
60000
80000
100000
120000
n input size

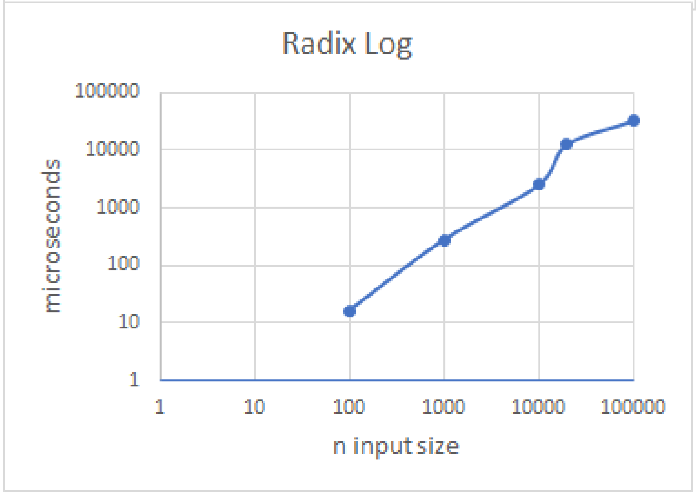

Bubble Log
microseconds
100000000
1000000
10000
100
1
1
10
100
1000
10000
100000
n input size

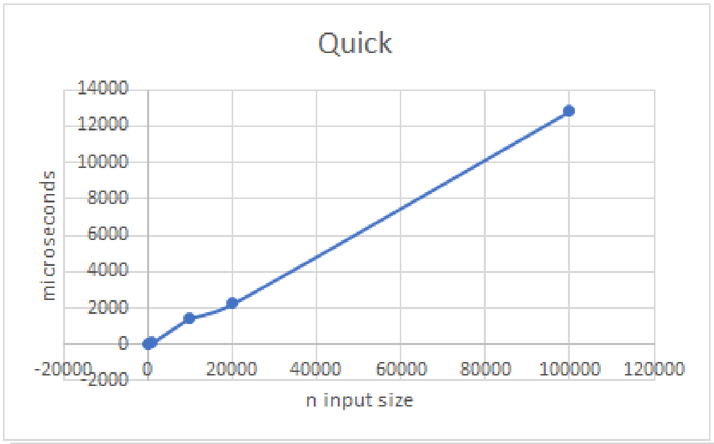

Counting Log
microseconds
100000
10000
1000
100
10
1
1
10
100
1000
10000
100000
n input size

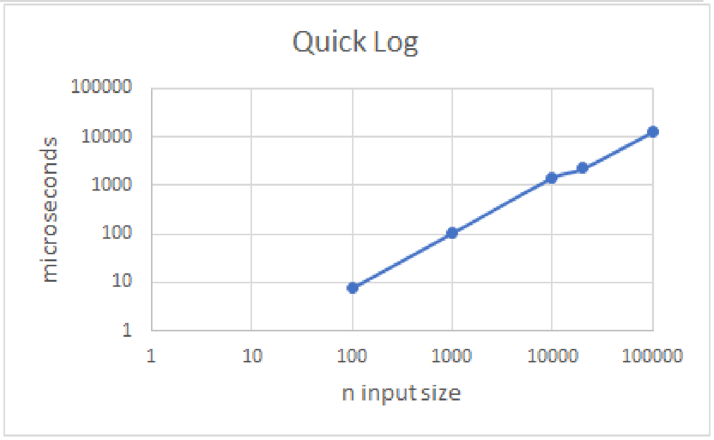

Exchange
microseconds
25000000
20000000
15000000
10000000
5000000
0
-5000000
-20000
0
20000
40000
60000
80000
100000
120000
n input size

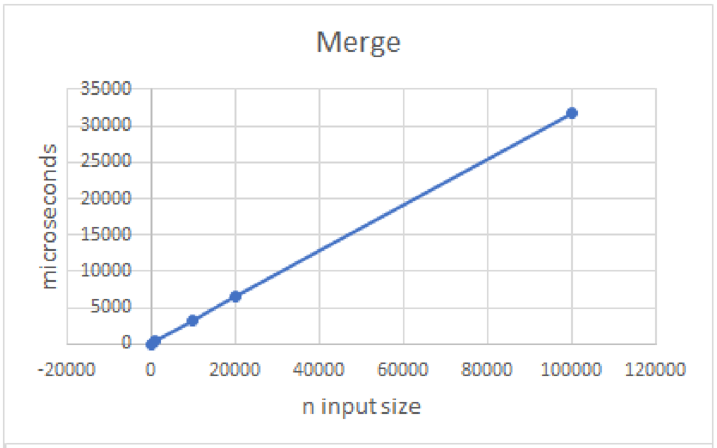

Radix
microseconds
35000
30000
25000
20000
15000
10000
5000
0
-20000
0
20000
40000
60000
80000
100000
120000
n input size

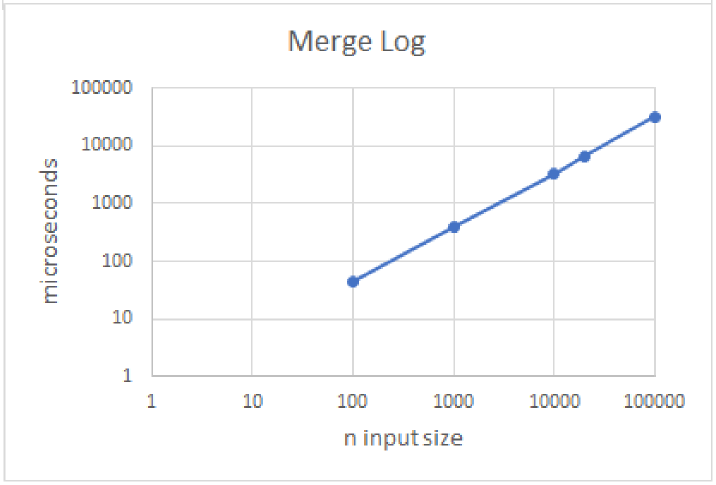

Exchange Log
microseconds
100000000
10000000
1000000
100000
10000
1000
100
10
1
1
10
100
1000
10000
100000
n input size


Radix Log
microseconds
100000
10000
1000
100
10
1
1
10
100
1000
10000
100000
n input size


Quick
microseconds
14000
12000
10000
8000
6000
4000
2000
0
-2000
-20000
0
20000
40000
60000
80000
100000
120000
n input size


Quick Log
microseconds
100000
10000
1000
100
10
1
1
10
100
1000
10000
100000
n input size


Merge
microseconds
35000
30000
25000
20000
15000
10000
5000
0
-20000
0
20000
40000
60000
80000
100000
120000
n input size


Merge Log
microseconds
100000
10000
1000
100
10
1
1
10
100
1000
10000
100000
n input size

Discussion and Conclusion:

Bubble sort and exchange sort both compare well with the theoretical complexity of $O(n^2)$. Both of the plots may appear somewhat linear in the non logarithmic plots; however, this is due partially to the fact that the input sizes jumped from twenty thousand to one hundred thousand. We ran some additional tests with other input sizes in between and the plots were visibly quadratic. Bubble sort and exchange sort both also performed at similar runtime speeds which aligns with the theoretical analysis. The simple sorts performed similarly to the other algorithms at smaller sizes but that performance quickly fell off as the input sizes grew.

Counting sort performed extremely well during our experimental analysis. Overall, it was the fastest sorting algorithm for the largest input size of 100,000 integers. This matches with the theoretical predictions of it being the fastest with a complexity of $O(n + k)$. One thing that the experimental results showed was that counting sort was performing faster with an input size of 100,000 than 20,000. This was due to a larger keyspace for the array of integers of size 20,000 which had a larger maximum integer that was much larger than the array containing 100,00 integers. In this case the k in the $O(n + k)$ played a big part in the runtime. Radix sort generally performed slower than counting sort, but not by a large amount. It is worth noting that radox is a good alternative but it is not that much slower and doesn't have as much added space in the memory. This data goes to show that deciding the correct algorithm really depends on the context of the problem and the distribution of the data.

Although both quicksort and mergesort have the same big O complexity of O(n*log(n)), , quicksort's constant is almost always smaller than mergesort's, resulting in quicksort performing faster. Quicksort compares less elements than merge sort, comparing all terms to the pivot element each time it is recursively called, resulting in one less comparison each iteration. Mergesort compares all items to each other, resulting in more comparisons. As the array goes further and further down, the time difference created by this is greater. Additionally, Quicksort does in-place operations, which does not require extra memory. On the other hand, mergesort makes two temporary arrays, resulting in operations not being in-place, and extra memory being allocated.