I. Insertion Sort

Insertion sort traverses through an array of size N starting with index 1 (instead of index 0) because in the case of the array only having a single value, it is sorted by default. The algorithm would them compare its' current index value to the previous values. If the current index value is less than the previous index values, they will be swapped. The algorithm continues to do this until the value is placed in the proper index before moving on to the next index and algorithm runs until every index in the array has been checked and placed in the proper position. This is considered a stable algorithm because if two or more indexes with the same value size are compared, they will not be swapped.

Time Complexity: $O(n^2)$

II. Merge Sort

The merge sort algorithm uses a divide and conquer strategy where the given array of size N is broken down into sub-arrays by splitting into two sub-arrays (labeled left and right). The algorithm uses recursive calling on the left sub-array followed by the right sub-array until every element is broken down to an array of a single element in which by definition it would be sorted. Once this process is completed, the arrays will merge together upon comparing index values and be sorted in their proper places when returning to the original array size. Because indexes of equal values do not swap, this algorithm is known to be stable as well.

Time Complexity: θ (nlogn)

III. Heap Sort

The heap sorting algorithm follows the heap properties where every parent is larger in value than its children. By definition, the root of the array or algorithm should be the element with the largest value. The algorithm would first build the array based on heap properties by checking to make sure the children are not larger than the root and then their parent node. In this algorithm we will be building a maxheap. For the sorting process, the last element and the first element of the array will be swapped which will break the heap properties. By default, the last element swapped would be sorted and therefore excluded from further swapping when recursion occurs. As for the element that got pushed to the root of the heap, it will traverse down prioritizing swaps with the child containing a larger value up until its position repairs the heap properties. We follow these steps again swapping the root with second to last element (because the last element is already sorted) and so on and so forth up until every element is sorted. Unlike the other two sorting algorithms above, heapsort is not a stable algorithm because as we run through the steps, even though indexes may contain the same value, they may potentially be swapped in positioning throughout the process.

Time Complexity: θ (nlogn)

IV. Quick Sort

The quicksort algorithm takes an index in our given array and sets it to the pivot point for the process. All the other numbers will be used to compare with the pivot point so elements less than the pivot value will reside on the left side of the array while elements larger will exist on the right side. In this scenario, the first element of the provided array will be our pivot point. Once the main array has been sorted out using this function, quick sort will be called recursively to handle the left side of the pivot followed by the right side using the first elements of both sub-arrays as the pivot point until everything is sorted into their proper positions. This algorithm is not stable because there is a possibility that adjacent elements with the same value could swap places throughout the process.

Time Complexity (Worst Case): O(n^2) * occurs when the array is reversely sorted and the pivot is set as the first or last element of the array*

Time Complexity (Average Case): O(nlogn)

V. Quick Sort (Random Pivot)

This version of the quicksort algorithm is exactly the same as the quicksort algorithm above with one difference; the pivot point is selected at random as any point of the given array which increases the chances of preventing the worst-case time complexity O(n^2) from occurring often.

Time Complexity (Worst Case): O(n^2) * Less chance of occurring if the pivot is not set to the first or last index of the given array [not impossible] *

Time Complexity (Average Case): O(nlogn)

Compile Test Times

Array Size: 10 (Milliseconds)	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Quick Sort (Random Pivot)
Trial 1	0.111328125	0.245117188	0.189941406	0.138916016	0.179931641
Trial 2	0.104003906	0.192138672	0.188964844	0.135986328	0.177978516
Trial 3	0.103027344	0.188720703	0.182861328	0.145996094	0.183105469
Trial 4	0.111328125	0.182128906	0.185058594	0.138671875	0.260986328
Trial 5	0.102294922	0.190917969	0.194091797	0.143798828	0.184814453
Trial 6	0.109863281	0.183105469	0.185058594	0.157714844	0.259765625
Trial 7	0.106933594	0.222900391	0.198242188	0.142822266	0.177001953
Trial 8	0.105957031	0.181152344	0.186035156	0.138183594	0.187988281
Trial 9	0.101806641	0.273925781	0.186035156	0.145996094	0.192626953
Trial 10	0.100097656	0.181884766	0.190917969	0.266113281	0.186279297
Average Time:	**0.105664063**	**0.204199219**	**0.188720703**	**0.155419922**	**0.199047852**

Analysis: Based on this data gathered, all the algorithms seem to sort at relatively the same amount of time without any noticeable difference.

Array Size: 100 (Milliseconds)	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Quick Sort (Random Pivot)
Trial 1	1.662841797	14.90307617	17.56738281	16.77514648	10.77197266
Trial 2	1.583007813	14.40405273	18.12597656	16.66699219	10.78979492
Trial 3	1.864013672	15.07275391	19.64501953	16.94604492	11.10913086
Trial 4	1.646972656	14.24291992	18.4519043	17.80419922	11.69213867
Trial 5	1.654785156	15.65087891	17.31005859	16.90771484	14.70678711
Trial 6	1.691894531	14.89404297	17.8371582	16.34008789	10.88916016
Trial 7	1.675048828	16.9909668	17.81616211	16.46118164	10.84423828
Trial 8	1.70703125	14.28637695	17.33374023	16.08764648	10.88696289
Trial 9	1.736083984	14.60717773	17.4909668	16.65991211	13.0871582
Trial 10	1.696777344	14.67578125	18.00097656	16.8503418	11.62280273
Average Time:	**1.691845703**	**14.97280273**	**17.95793457**	**16.74992676**	**11.64001465**

Analysis: Based on the data, insertion sort is still superior to the other sorting algorithms even though it has the worst time complexity of $\mathbf{O(n^2)}$ while the other algorithms all have an (average) time complexity of $\mathbf{\theta(nlogn)}$. As for the quick sort algorithms, between default and random pivot, running with a random pivot definitely makes a big difference in terms of processing time.

Array Size: 1000 (Milliseconds)	**Insertion Sort**	**Merge Sort**	**Heap Sort**	**Quick Sort**	**Quick Sort (Random Pivot)**
Trial 1	170.4597168	1990.460938	2820.51416	11891.94287	1397.207031
Trial 2	171.7021484	2053.054932	2821.127686	12189.4751	1479.37915
Trial 3	167.223877	2002.508301	3176.853027	12142.95801	1427.541992
Trial 4	168.7241211	1992.28418	3106.721924	12030.12183	1418.905029
Trial 5	181.3459473	1980.428955	3014.813965	12607.87915	1387.511963
Trial 6	169.9711914	2060.838623	2995.588623	12383.22607	1437.447998
Trial 7	168.2297363	1964.310791	3065.456787	12585.24219	1411.529053
Trial 8	165.9250488	1981.244873	3149.943115	12217.5249	1404.902832
Trial 9	171.3122559	2012.381104	3027.135986	12356.25195	1421.0271
Trial 10	165.2529297	1951.037109	3041.237061	12171.74902	1432.190918
Average Time:	**170.0146973**	**1998.85498**	**3021.939233**	**12257.63711**	**1421.764307**

Analysis: With the exclusion of insertion sort algorithm, quick sort at a random pivot overall handles large sized arrays better than the other algorithms followed by merge and heap sort even though merge, heap and quick sort had relatively the same theoretical running time.

Compiling Errors:

- Slightly concerned about as to why insertion sort processed the fastest even though it has the worst time complexity out of the bunch
- Running each algorithm in an array sized 10,000+ took a relatively long time (ten or more minutes) which was too time consuming to gather data for
- As for array size of one million, compiler stops completely