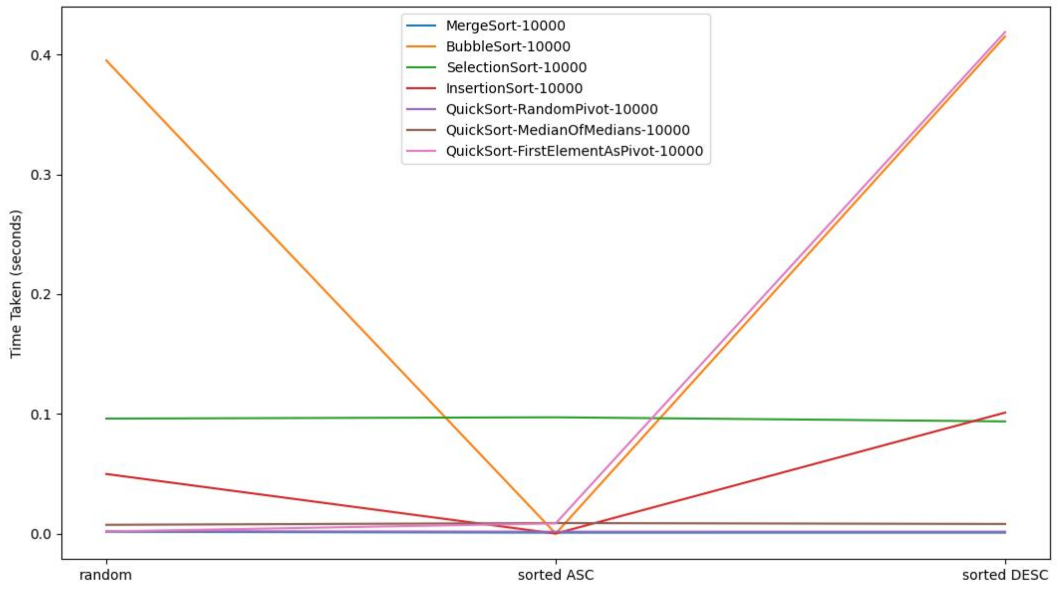
How to run

- **Ubuntu 22.04**

```
make setup-ubuntu-focal # installs g++ python3 and python3-matplotlib packages
make                    # compile
make run                # generates the plot
```

Sorting Algorithm Time Complexities

Sorting Algorithms Time taken(in seconds) Comparison when the given array is Unsorted, Sorted in Ascending Order and Sorted in Descending Order with an array of size 10000 containing random numbers



Sorting Algorithm(Array size: 10000)	Unsorted Array	Sorted Array (Asc)	Sorted Array (Desc)
MergeSort	0.00156972	0.000814442	0.000831642
BubbleSort	0.395118	0.0000174	0.415203
SelectionSort	0.0960688	0.0971576	0.0937215
InsertionSort	0.0497988	0.00002786	0.101039
QuickSort (Random Pivot)	0.00209612	0.00159786	0.0016165
QuickSort (Median of Medians)	0.0073482	0.00885023	0.00806218
QuickSort (First Element as Pivot)	0.00182434	0.00855287	0.41885

Bubble Sort

- **Best Case**: Occurs when the input array is already sorted. In this case, the algorithm only requires a single pass to determine that no swaps are needed. This can be seen from the above figure for the sorted array case.
- **Worst Case**: Occurs when the input array is in reverse order. This results in the maximum number of comparisons and swaps in each pass. This can be seen from the above figure for the array sorted in descending order case.
- **Average Case**: Occurs when the input data is random or partially sorted, resulting in an average number of comparisons and swaps. This can be seen from the above figure for the unsorted array case.

Selection Sort

- **Best Case**: There is no distinction in the number of comparisons, regardless of the input data. Selection sort always performs the same number of comparisons, leading to a time complexity of O(n^2). This can be seen from the above figure.
- **Worst Case**: There is no distinction in the number of comparisons, regardless of the input data. Selection sort always performs the same number of comparisons, leading to a time complexity of O(n^2). This can be seen from the above figure.

- **Average Case**: Same as the worst case since it always performs the same number of comparisons. Selection sort always performs the same number of comparisons, leading to a time complexity of O(n^2). This can be seen from the above figure.

Insertion Sort

- **Best Case**: Occurs when the input array is already sorted, requiring minimal comparisons and no swaps. This can be seen from the above figure for the sorted array case.
- **Worst Case**: Occurs when the input array is in reverse order, resulting in the maximum number of comparisons and swaps in each pass. This can be seen from the above figure for the array sorted in descending order case.
- **Average Case**: Occurs when the input data is random or partially sorted, leading to an average number of comparisons and swaps. This can be seen from the above figure for the unsorted array case.

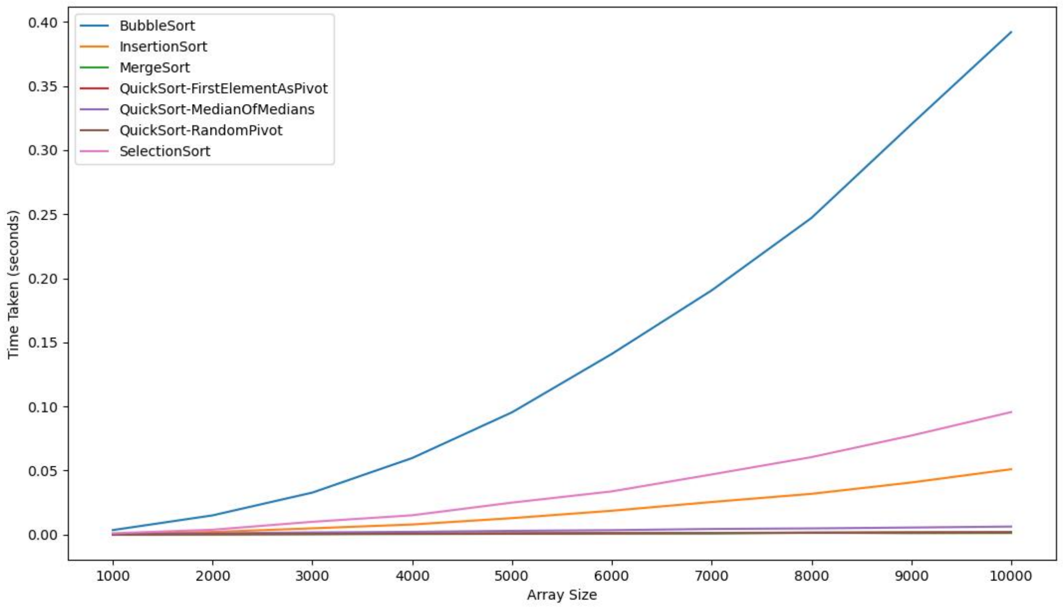
Quick Sort

- **Best Case**: Occurs when the pivot choices consistently divide the input into nearly equal partitions, resulting in O(n log n) time complexity. This can be seen for both quicksort with median of medians and quicksort with random pivot although quicksort with random pivot is almost always slightly better.
- **Worst Case**: Occurs when the pivot choices result in highly unbalanced partitions, such as always picking the smallest or largest element as the pivot. This leads to a time complexity of O(n^2). Can be seen in QuickSort (First Element as Pivot) for elements sorted in descending order case.
- **Average Case**: Occurs when the pivot choices are relatively balanced, leading to an average time complexity of O(n log n). This can be seen for both quicksort with median of medians and quicksort with random pivot although quicksort with random pivot is almost always slightly better.

Merge Sort

- **Best Case**: The best-case scenario for Merge Sort is when the input array is already sorted. In this case, Merge Sort still divides the array into sub-arrays and merges them, but the merging step is more efficient because it involves fewer comparisons. Therefore, the best-case time complexity is O(n log n), which is the same as the average and worst-case time complexity.
- **Worst Case**: The worst-case time complexity of Merge Sort is also O(n log n). The worst-case scenario occurs when the input array is in reverse order. In this case, Merge Sort still divides the array into sub-arrays and merges them, but it involves the maximum number of comparisons and operations.
- **Average Case**: The average-case time complexity of Merge Sort is O(n log n). This is because, on average, the algorithm divides the array into roughly log(n) levels of sub-arrays, and at each level, it performs n operations to merge the sub-arrays. **In practice the time difference due to comparisons becomes insignificant as can be seen from the above graph

Sorting Algorithms Time taken(in seconds) Comparison with unsorted arrays of multiple sizes containing random numbers



Array Size	BubbleSort	InsertionSort	MergeSort	QuickSort-FirstElementAsPivot	QuickSort-MedianOfMedians	QuickSort-RandomPivot	SelectionSort
1000	0.00353063	0.000537021	0.000145	0.00014186	0.000485461	0.00016576	0.00103414
2000	0.0150297	0.00205924	0.000242981	0.000301221	0.00102046	0.000344561	0.00381413

Array Size	BubbleSort	InsertionSort	MergeSort	QuickSort-FirstElementAsPivot	QuickSort-MedianOfMedians	QuickSort-RandomPivot	SelectionSort
3000	0.0328166	0.00492221	0.000344761	0.000502441	0.00171066	0.000559641	0.0100133
4000	0.0597726	0.00790258	0.000472201	0.000712581	0.00219084	0.000773381	0.0150913
5000	0.0954308	0.0128629	0.000629721	0.000884802	0.00293433	0.00100306	0.0249843
6000	0.140983	0.0186286	0.000944822	0.0010564	0.00346695	0.00114328	0.0337484
7000	0.19053	0.0254442	0.000872162	0.00132962	0.00440549	0.00134946	0.0469624
8000	0.247033	0.0318388	0.00148876	0.001412	0.00486321	0.00158292	0.0604736
9000	0.320001	0.0406818	0.0011816	0.0017195	0.00553061	0.00178866	0.0772598
10000	0.391965	0.0510277	0.00130014	0.00187294	0.00629449	0.00201064	0.095597

Bubble, Selection, and Insertion Sort

- Bubble, Selection, and Insertion Sort algorithms have a much steeper increase in execution time as the array size grows. This behavior is expected as these algorithms have a time complexity of O(n^2), where the execution time increases quadratically with the array size.
- For each of these algorithms, the execution time increases significantly as the array size goes from 1000 to 10000, indicating their poor scalability with larger datasets.

MergeSort and QuickSort

- MergeSort and various QuickSort variants (Random Pivot, First Element as Pivot) exhibit significantly better performance as the array size increases compared to Bubble, Selection, and Insertion Sort.
- These algorithms have an average-case time complexity of O(n log n), which makes them more efficient for larger dataset sizes.

QuickSort Variants

- The data shows that the QuickSort variants, including Random Pivot and First Element as Pivot, perform quite similarly and are consistently faster than Bubble, Selection, and Insertion Sort. This demonstrates the effectiveness of the QuickSort algorithm for sorting arrays in practice.

Median of Medians QuickSort

- The Median of Medians QuickSort variant performs slightly worse than the other QuickSort variants. This could be due to the additional overhead of selecting the median of medians as the pivot, which might not always provide significant benefits in practice, especially for small to moderately sized datasets.

MergeSort vs. QuickSort

- Both MergeSort and QuickSort variants show similar performance across the dataset sizes. This indicates that their average-case time complexities are indeed quite close, despite QuickSort being considered a bit more efficient in practice for many cases. The differences in execution time could be attributed to various factors, such as pivot selection and partitioning strategies.

In summary, the provided data clearly demonstrates the importance of algorithm choice in sorting tasks. Bubble, Selection, and Insertion Sort algorithms exhibit poor scalability with larger datasets due to their O(n^2) time complexity, while MergeSort and QuickSort variants with an average-case O(n log n) time complexity provide much better performance. Additionally, the data suggests that the Median of Medians QuickSort variant may not always offer a significant advantage over other QuickSort strategies in practical scenarios.