Ali Sultan Sikandar & Shakh Saidov

3/20/2020

Project 1 Report

Introduction

The purpose of the project is to investigate different sorting algorithms and analyze their running time. Sorting algorithms differ from each other in that they may be more efficient with a certain pattern within the list than for others. This is analyzed in this project by calculating the running time for sorting for the algorithms - namely Bubble Sort, Insertion Sort, Merge Sort, Selection Sort, QuickSort with 1st element as the pivot, QuickSort with the median element of the first,middle and last element as the pivot, Quicksort by using a random element as the pivot, Java Library Merge Sort and Java Library Quicksort. The running time for different numbers of elements, ranging from 0 to 100000, is calculated for the quadratic sorts and elements ranging from 0 to 95000000 are sorted for the rest of them. The results are plotted in a graph to give us a clear visual conclusion of the advantages and disadvantages of all of these algorithms when compared against each other.

Bubble Sort:

Bubble Sort sorts a list by identifying the largest element in the list by comparing every element next to each other and then moving (bubbling) the largest element to the end of the list. In theory the best running time complexity is given by big-O(n). This happens when the list is already sorted. Therefore, the more sorted the list is, the more efficient the bubble sort algorithm is. The worst case time complexity is big-O(n^2), which is also the average running time complexity of bubble sort. The only significant advantage bubble sort has over other algorithms is that it can detect a sorted list efficiently. However, it is a poor sorting algorithm when it comes to real life scenarios where you have to sort lists of big size.

Insertion Sort:

Insertion sort works by sorting the list one at a time. It passes through the entire list and sorts whatever it leaves behind. In theory the best running time complexity is given by big-O(n). This happens when the list is already sorted. Therefore, the more sorted the list is, the more efficient the insertion sort algorithm is. The worst case time complexity is big-O(n^2), which occurs when insertion sort is implemented on a reverse sorted array. The average case time complexity is also big-O(n^2). Having a low overhead, insertion sort is particularly useful while sorting lists of small sizes. However, similar to bubble sort, it is useless in real life scenarios while dealing with large lists.

Selection Sort:

Selection Sort algorithms sort a list by identifying the smallest element in a list in every pass and swapping them with elements in the start of the list in a sequential order. Since the algorithm always runs through the entire list n times, there exists no best, worst or average cases and the time complexity is always big-O(n^2). Selection sort is only beneficial when there's limited auxiliary space. It is disadvantageous because compared to other quadratic sorting algorithms, selection sort does not reap benefits in situations where the order of the list is a certain way.

Merge Sort:

Merge Sort algorithm sorts a list of elements by using the divide and conquer principle. It splits the list in halves recursively then sorts the individual pairs. After that the whole list is sorted when it builds back up. The time complexity of merge sort is big-O(nlgn) for the worst, best and average cases. Merge sort is a very efficient sorting algorithm when dealing with big lists, and uses almost constant time irrespective of the pattern within the list. It also preserves the original order and hence is regarded as stable. The only disadvantage merge sort has is that it requires extra space in the memory when splitting the array into half.

Quick Sort:

Quicksort operates by sorting a list using the partitioning. This is done by selecting one element as a pivot and then treating the elements before and after this pivot as two seperate lists and sorting them recursively. The choice of pivot is vital because the running time can be affected by it. The average running time and the best running time is big-O(nlgn). While it is common to use the first element as the pivot, if the list is already sorted, choice of first pivot causes the running time complexity to degrade to big-O(n^2). Therefore to eliminate it, the median of the first, last and middle element or just any random element is used as a pivot to achieve the best case of big-O(nlgn). Quicksort has many advantages: it doesnt use extra space like merge sort, is efficient for almost all patterns existing within the list if the right element is chosen as a pivot and works well with lists of large sizes. One of the disadvantages, however, is that quicksort is unstable, which means that the original order of the elements that may be similar is never preserved.

Approach

To conduct our experiments and expose the advantages and disadvantages of the various sorting algorithms, we used a set number of seeds for outputting random numbers. We used a different range for the number of elements for efficient and inefficient sorting algorithms.

All NumberOfInput used: 0, 50, 100, 500, 1000, 5000, 10000, 12000, 42000, 50000, 100000, 500000, 1000000, 10000000, 39000000, 95000000

For inefficient algorithms - Insertion Sort, Selection Sort, and Bubble Sort - we used input data ranging from 0 - 100,000 elements.

For efficient algorithms - Merge Sort and Quick Sort - we exceeded 10,000,000 but had to modify the exact ceiling at times, due to stack overflow errors occurring when numbers were too large.

For random-ordered arrays, we used 5 different seeds - 2468, 515, 121252, 6, 12521 - and filled the array with random elements.

For reverse-ordered arrays, we used a for-loop and iterated backwards from one less than the size of the input data down to 0. (Ex. if we were testing 10,000 elements, we filled the array with 9,999 as the first element and 0 as the last.)

For almost-sorted arrays, we decided to divide arrays up as 75% sorted and 25%unsorted, so inside a for-loop, we randomized every 4th element through the seeds we mentioned above.

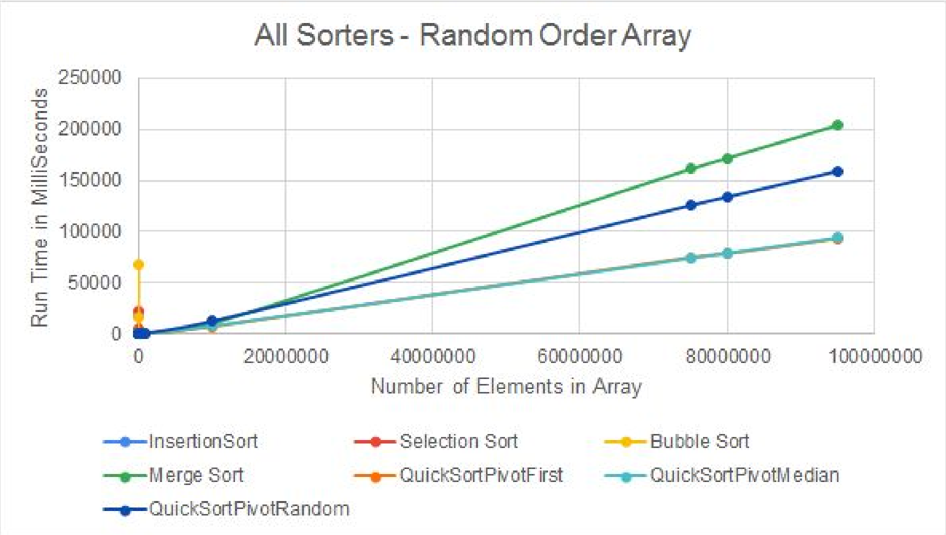
We used this particular approach because the properly spread out number of inputs helped us create a gradual graph that visually represents the difference between various sorting algorithms. The seeds we used were random but the number 5 seemed enough to conclude the average run time.

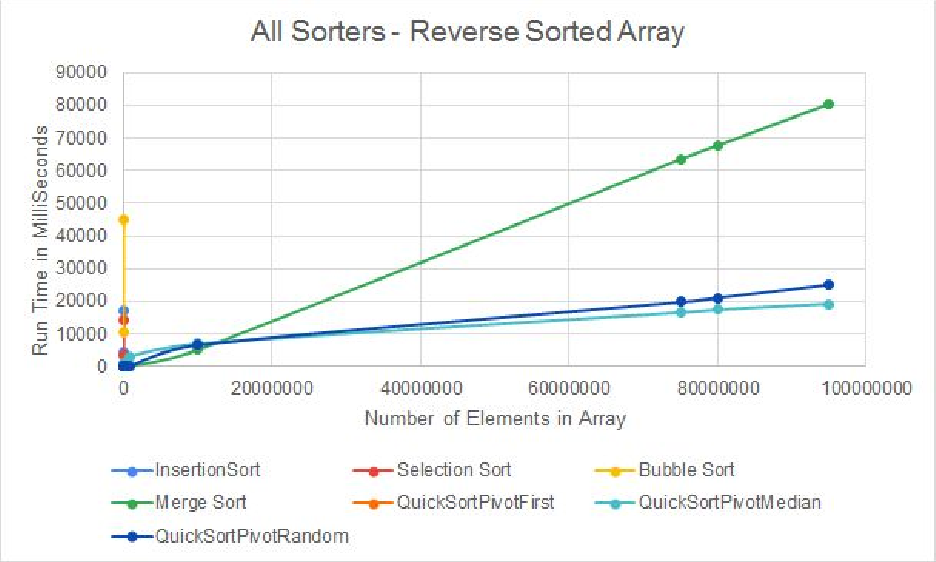
Methods

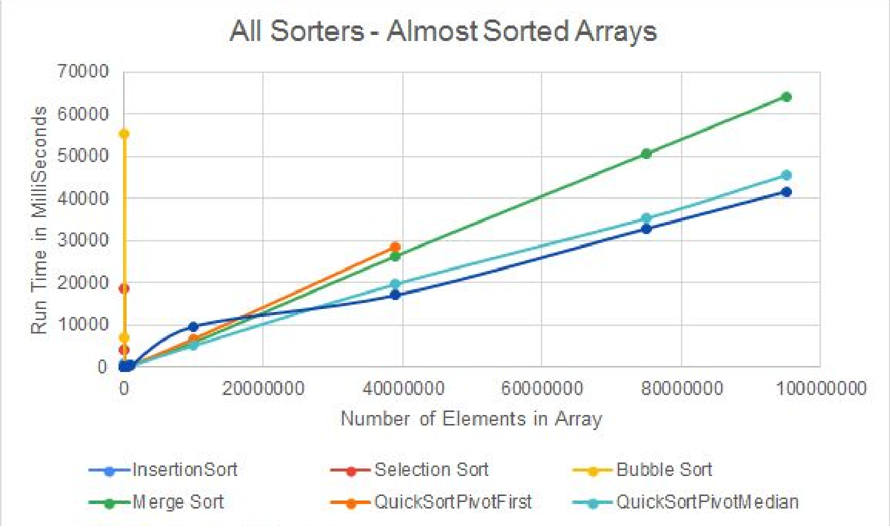
We used two different machines in our experimental setup - MacBook Air and a Chromebook. To calculate the run time of our sorting algorithms, we used the inbuilt Java method called “System.currentTimeMillis()”. We repeated each experiment 5 times, as we believed that is a considerable amount to come to a proper conclusion. The times reported are the average of 5 experiment-controller runs, shown in milliseconds.

Data & Analysis

After the end of our experiment, the resulting graphs looked as below. We compared the inefficient and efficient sorters separately, but when taken altogether, you can clearly see the difference between them.

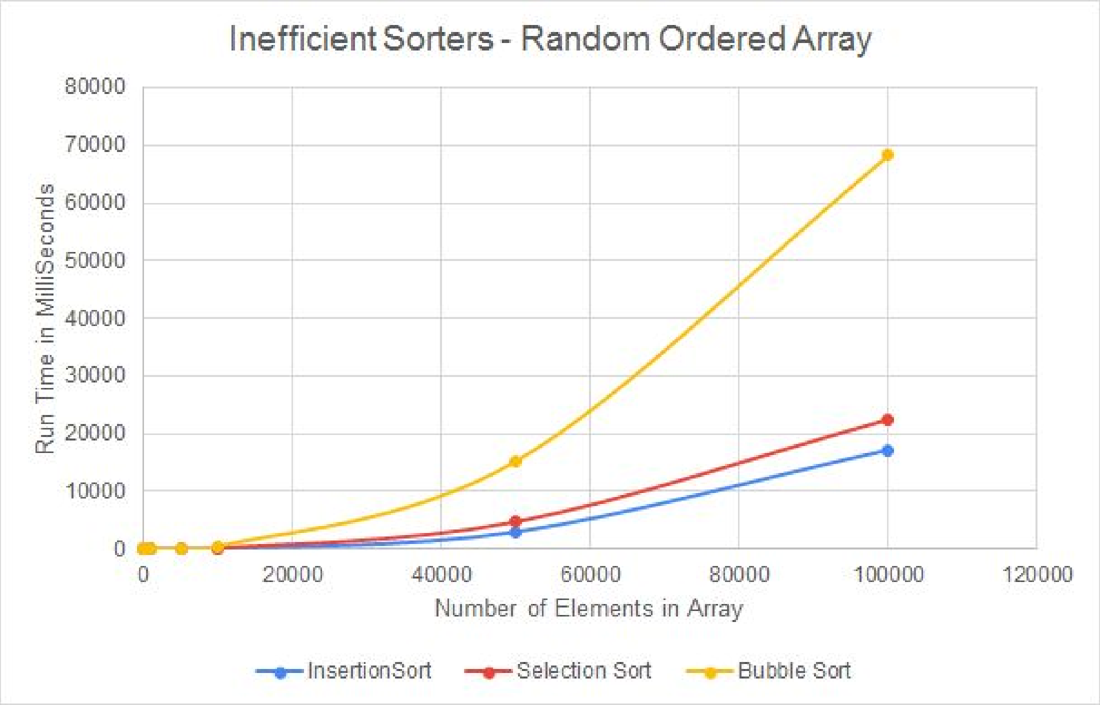




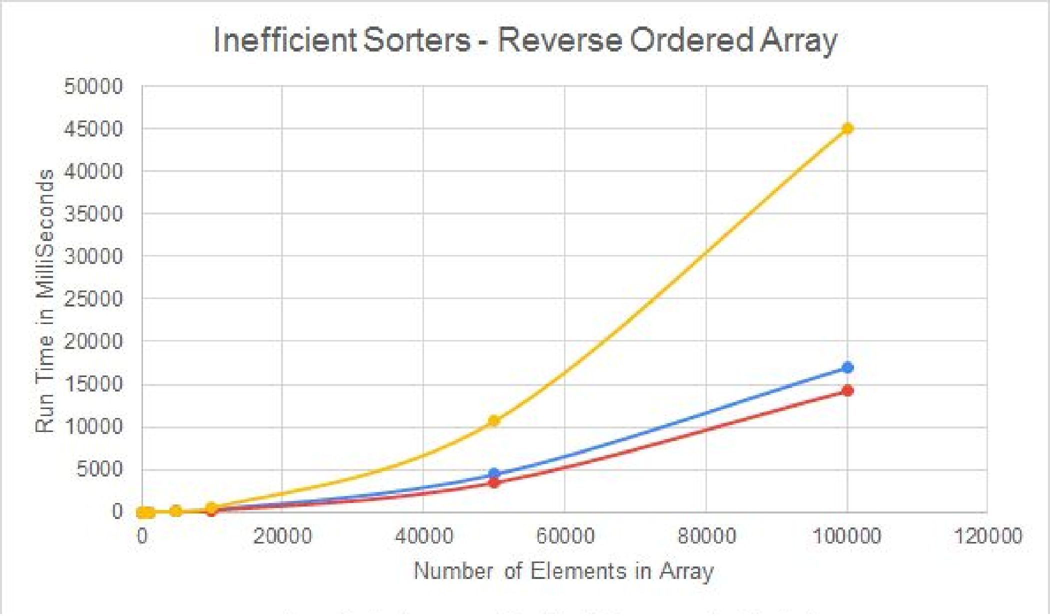


In the inefficient sorters - bubble sort, insertion sort and selection sort, the visible differences between the algorithms can be seen when they are implemented on arrays containing a large number of elements.

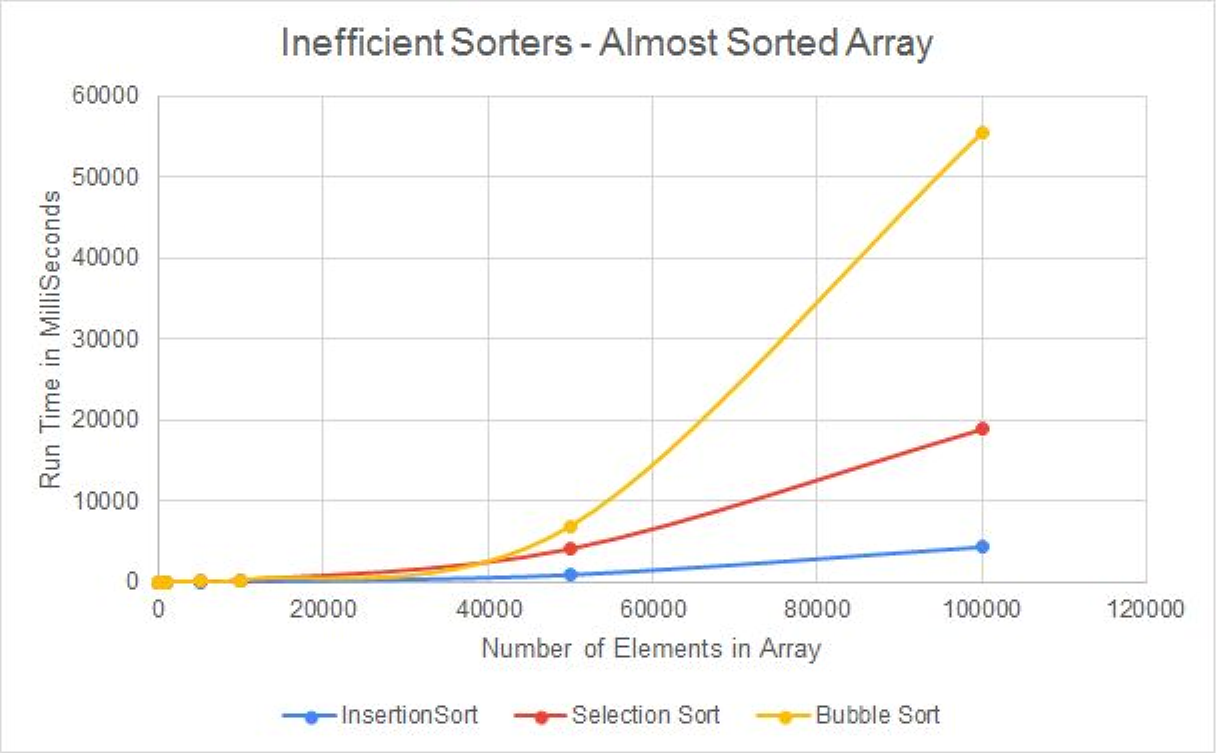
Bubble Sort is seen as the worst of the three in all three arrays (random ordered, almost sorted and reverse sorted)



We see that selection sort performs slightly better than insertion sort (despite insertion sort being an overall better algorithm of the two) in the case below because a reverse sorted array is the worst case for the insertion sort.



An already sorted array, or something similar to it is theoretically favorable for bubble sort and insertion sort. As expected, Insertion sort performs excellently in this case as expected for all numbers of elements. While bubble sort also follows the same pattern (i.e excellent performance for almost sorted) for smaller arrays, when the number of elements exceeds 6000, it performs better with a reverse sorted array in comparison to the almost sorted one.

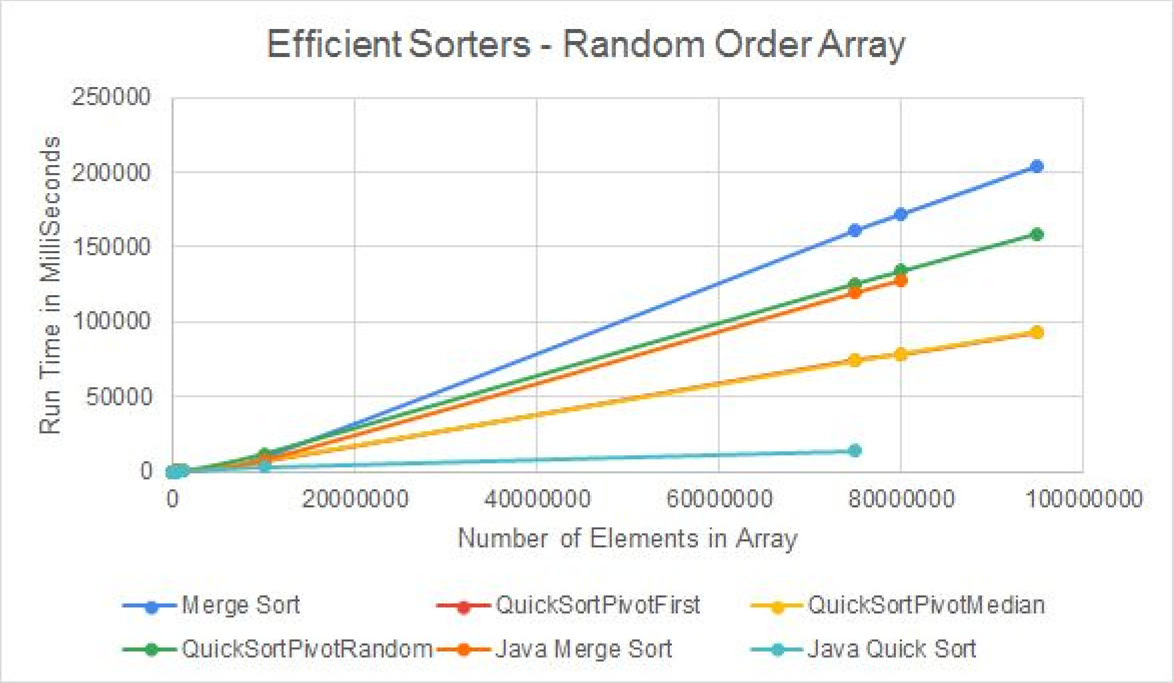


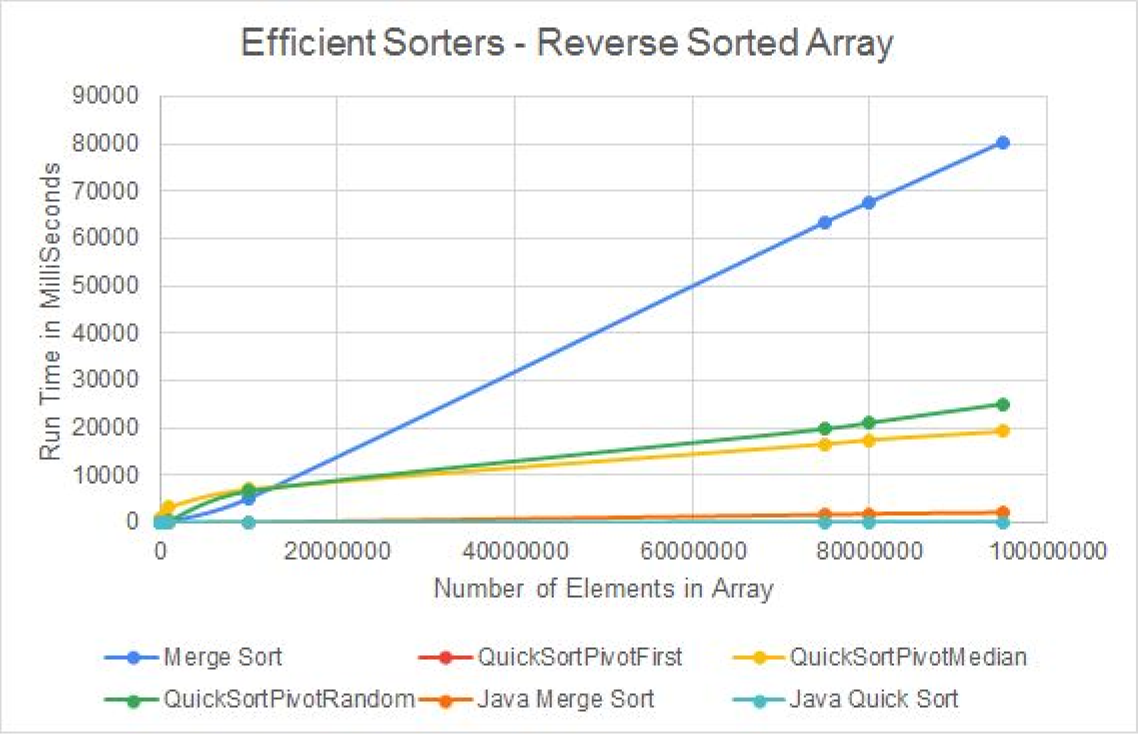
We can see that the graphs of efficient sorters differ from the inefficient quadratic sorters in that they display somewhat of a linear relationship as theoretically expected.

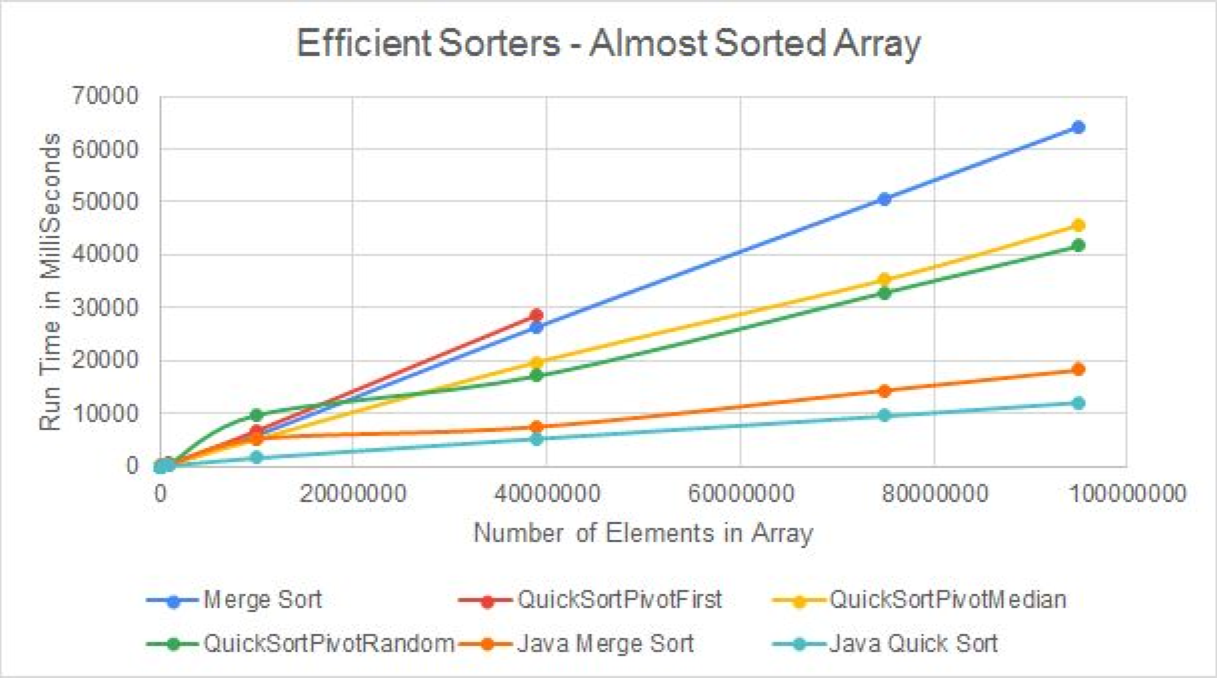
In all the efficient sorters, we see that to obtain better and consistent results, the choice of pivot as the median is always more favorable compared to other choices. As expected, the choice of first element as the pivot for an almost sorted array gives us the worst results. In addition to that, choice of the first element as the pivot can cause the program to encounter a stack overflow error. The choice of a Random element as the pivot is good since you encounter no error at all, but the running time is varying and hence not reliable - it may perform better than the other pivot choices or it may perform worse (as is seen in the case of the almost sorted array where pivotRandom works better than pivotMedian). Therefore, the choice of Median pivot is the best way to go.

Merge Sort performs worse than all the quicksort versions, but it is still significantly better than all the inefficient sorting algorithms. We also conclude that merge sort performs better with a random ordered array than for reverse or almost sorted array.

The inbuilt Java Merge sorter and Quick Sorter, however, yields the best result when compared to our implementation of the Merge Sort and Quick Sort, as can be seen from the graphs.




Efficient Sorters - Reverse Sorted Array
Run Time in MilliSeconds
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0
20000000
40000000
60000000
80000000
100000000
Number of Elements in Array
Merge Sort
QuickSortPivotFirst
QuickSortPivotMedian
QuickSortPivotRandom
Java Merge Sort
Java Quick Sort


Efficient Sorters - Almost Sorted Array
Run Time in MilliSeconds
70000
60000
50000
40000
30000
20000
10000
0
0
20000000
40000000
60000000
80000000
100000000
Number of Elements in Array
Merge Sort
QuickSortPivotFirst
QuickSortPivotMedian
QuickSortPivotRandom
Java Merge Sort
Java Quick Sort

Conclusion

After conducting our experiment, we came to the conclusion that inefficient sorting algorithms, such as Bubble Sort, Insertion Sort and Selection Sort, aren't useful when the number of data exceeds 100,000. If you were to use these sorts, Selection and Insertion Sort would be advised as Bubble Sort performs the worst in all cases.

Out of the efficient sorting algorithms, QuickSort (with a median pivot) would be the preferred option. Note, Java Library's implementation of quicksort seems the most optimal.

Thus, we learned the distinguishing factors between sorting algorithms and how the implementation of a method affects its efficiency.

References

- System. (2018, October 6). Retrieved February 28, 2020, from https://docs.oracle.com/javase/7/docs/api/java/lang/System.html
- Random. (2018, October 6). Retrieved February 28, 2020, from https://docs.oracle.com/javase/7/docs/api/java/lang/System.html
- Weiss, M. A. (2010). *Data structures & problem solving using Java*. Boston, MA: Pearson.
- Lakshay Gupta. (N.d.) *QuickSort.* Geeksforgeeks. Retrieved March 1, from https://www.geeksforgeeks.org/quick-sort/
- Abhishek Tiwari(N.d.) *Median.* Geeksforgeeks. Retrieved March 6, from https://www.geeksforgeeks.org/median/
- Ge Xia. (N.d.) *Lecture Note 2.* Moodle. Retrieved March 4, from https://moodle.lafayette.edu/pluginfile.php/551806/mod_resource/content/1/sorting.pdf