Jxh604
EECS 233 Project 4 Report

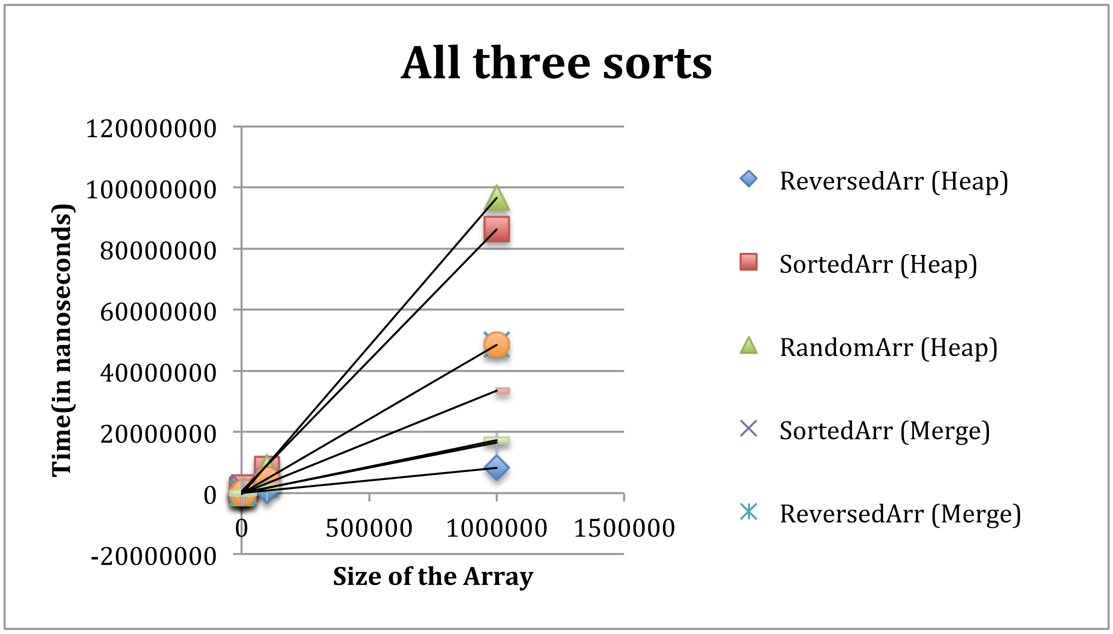
Heap Sort, Quick Sort, Merge Sort

NOTE: ALL OF THE DATA USED IN THIS REPORT IS COLLECTED FROM Report1HeapSortData.txt, Report1MergeSortData.txt, Report1QuickSortData.txt, and Sorts Data.xlsx. I know that the data is reliable because the trends for the curves used in the analysis below has an R^2 value of 0.9999 which is very close to 1, meaning that the curve is reliable. Furthermore, in the excel file, below every set of collected data, I have included the average and the variance, to show that my data is reliable.

Jxh604

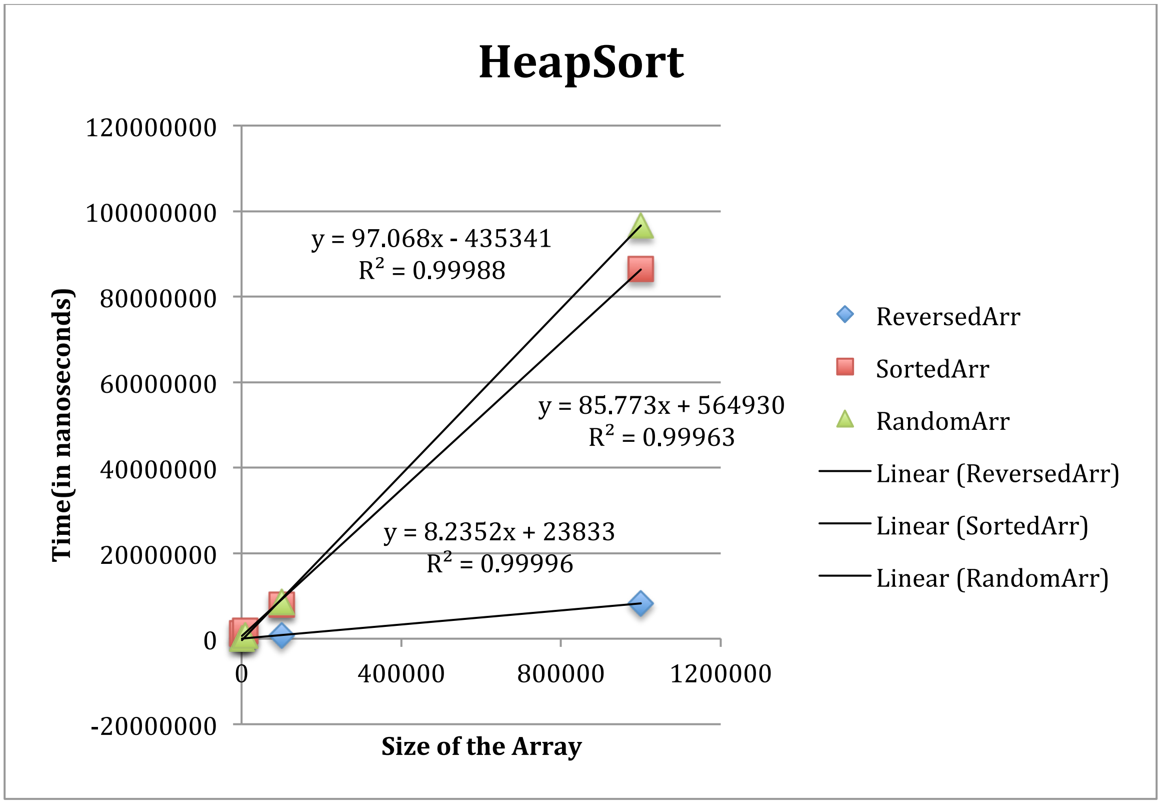
Jxh604
EECS 233 Project 4 Report

Summary:
Theoretically speaking, heap sort is the best algorithm to use, since it has a O(nlogn) for all cases, and uses O(1) for extra space. But based on my data, I cannot say that heap sort is the best sort for all cases. From the graph titled “All three sorts” we see that heap sort for random array has the highest length of time in nanoseconds, while having the best time for reversed array. From a practical stance, sorting algorithms will be applied to arrays and sets of data that are more often than others not sorted. So using the random array an indication of its effectiveness is acceptable. Because heap sort has the longest time for random array, I do not suggest the use of heap sort. Since it is also not stable, and requires more extra space than quick sort and merge sort. Continuing on to quick sort. I think that quick sort is great for sorting integers as we are doing here, since it has about the same complexity for sorted array, reversed array, and random array, and sorts in place. Merge sort takes less time than heap sort for random data but the down side of merge sort is that it takes O(n) space. When you are sorting 1000 elements, it may not be a lot, but sorting millions, trillions of elements will add up. In conclusion, I recommend using quick sort for sorting integers, because it sorts in place, takes up O(logn) extra space, and sorts the random array in the least amount of time for the random array compared to heap sort and merge sort. Sorting random arrays will be the most common case, and I recommend using quick sort for all sorts, because quick sort gives the best performance.



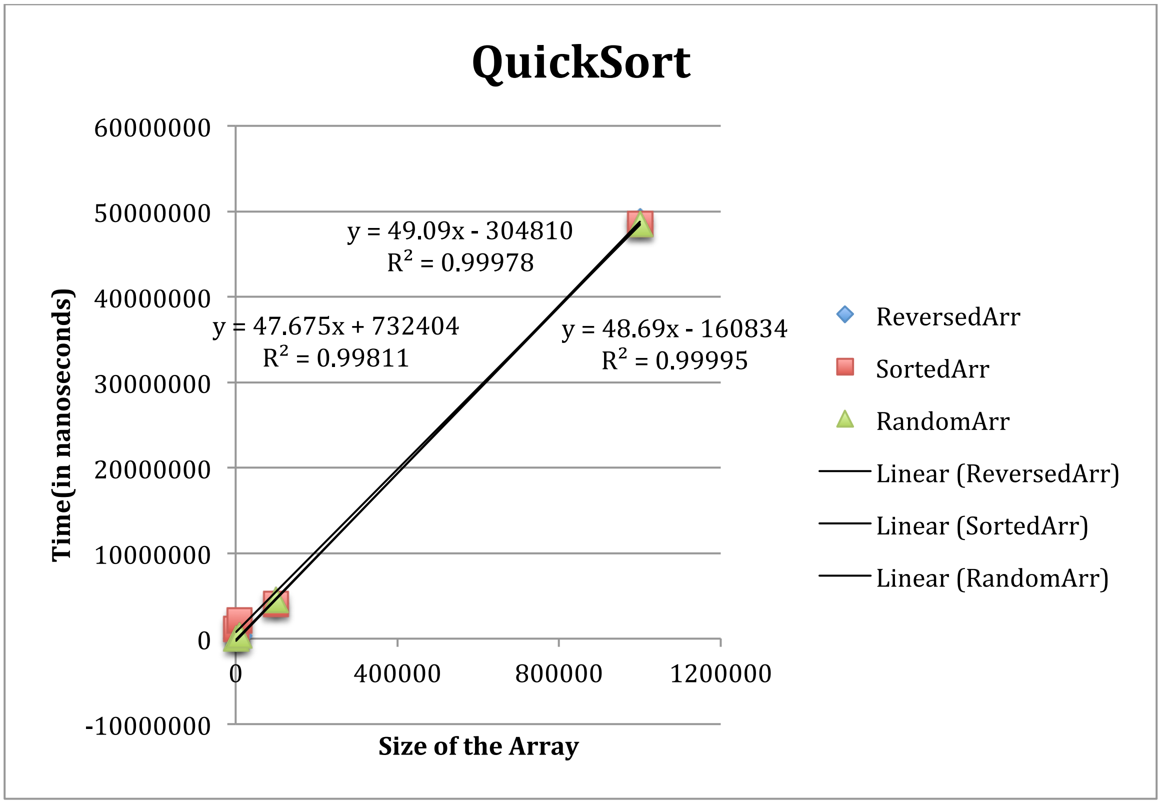
Heap Sort:
According to my data, heap sort’s best case is the reversed array, while the sorted array was slightly better than the random array. This makes sense, because the heap

is first in first out, and with my implementation of heap sort, the greater numbers would be popped first. Although the heap sort theoretically have a best, and average case complexity of O(nlogn), I still don't recommend using heap sort, because maintaining the heap is costly, especially for a large sized array. According to the graph below, as well as the data collected from Report1.java that is analyzed in Sorts Data.xlsx that is included in this zip file, I see a dramatic increase in the time taken due to the increase in size of the array. For example, in the reversed array for heap sort, for the 1000 element array, it took 64000 nanoseconds to complete the sort, whereas it took 8259840 to complete the sort for 1000000-element array. Also, a pit fall of heap sort is that it does not differentiate between two same elements. In this case, we are comparing integers to one another, but in another event that a name and an employee number represent each element, and we sort the names first, and the numbers second, we expect the employee numbers to be sorted within the employees who have the same name, but we are not able to differentiate between two people who happens to have the same name because heap sort's pitfall lies here.



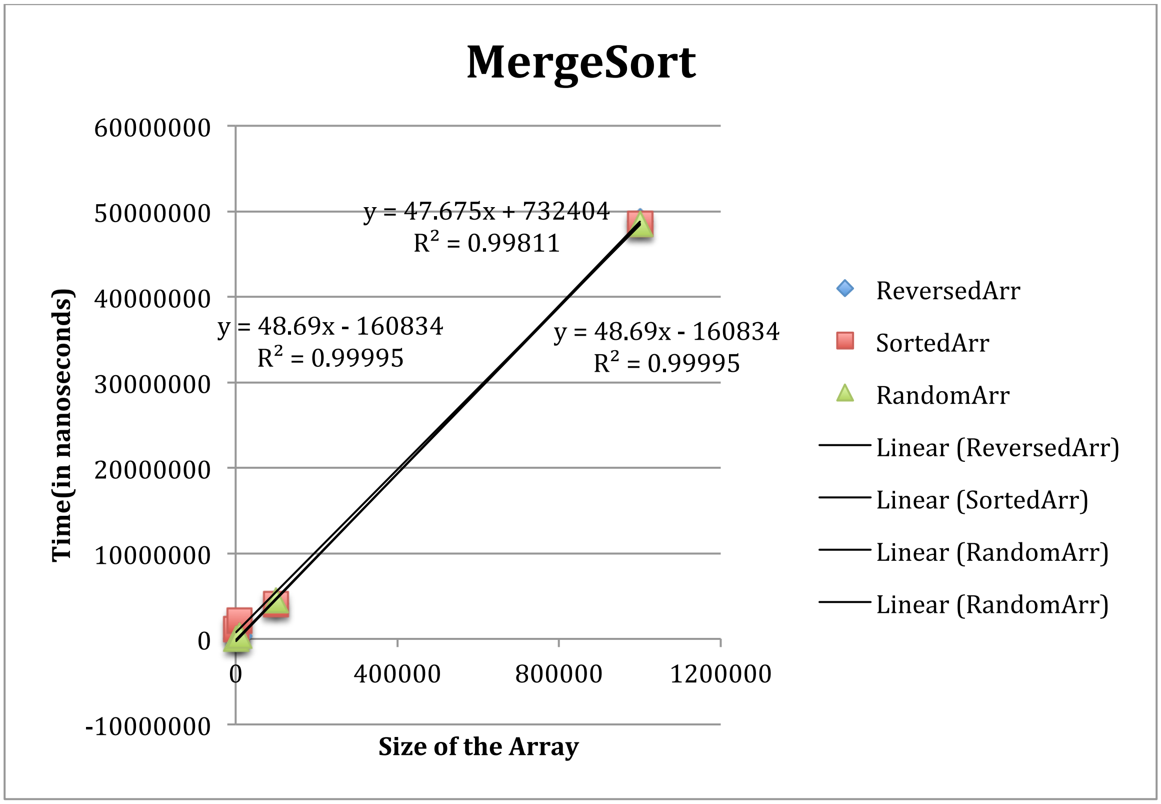
Quick Sort:

According to my data, quick sort is relatively the same for all three arrays, sorted array, inversed array, and random array. Quick sort is advantageous in that it is very space efficient, it sorts in place, requiring O(logn) extra space. The downside of quick sort like heap sort is that it is not stable. In this case, we are comparing integers to one another, but in another event that a name and an employee number represent each element, and we sort the names first, and the numbers second, we expect the employee numbers to be sorted within the employees who have the same name, but we are not able to differentiate between two people who happens to have the same name because heap sort's pitfall lies here.



Merge Sort:
Very fast, but requires space and have a O(n) for extra space. Merge sort also performs similarly among the sorted array, reversed array, and the random array. This is because merge sort breaks down the array to size 1 and merge them individually back together even if the array is sorted, not sorted, or random. The pitfall that I see for merge sort is that it takes up space. But performance wise, it is very efficient.

Jxh604
EECS 233 Project 4 Report


MergeSort
Time(in nanoseconds)
Size of the Array
y = 47.675x + 732404
R² = 0.99811
y = 48.69x - 160834
R² = 0.99995
y = 48.69x - 160834
R² = 0.99995
ReversedArr
SortedArr
RandomArr
Linear (ReversedArr)
Linear (SortedArr)
Linear (RandomArr)
Linear (RandomArr)