Practical 6 – Efficent Sorts (Quick Sort)

30/3/2021

Q1. Implement Quick Sort using pseudo code.

```
    private static void partition(int[] a, int low, int high){
        int l = low; // low is index 0th
        int h = high; // high is the last index
//        int pivot = a[high-1];
        int pivot = a[low+(high-low)/2];
        while(l<=h){


            while(a[l] < pivot) // as long as the left index is smaller
than the median, increment rightwards
                l++;
            while(a[h]>pivot) // as long as the rightmost element is
greater than the median, decrement leftwards
                h--;
            if (l<=h){
                helperSwap(a, l, h);
                l++; // moves both sides to the next index
                h--;
            }
        }
//          calls partition method recursively
        if (low<h)
            partition(a, low, h);
        if (l<high)
            partition(a, l, high);
    }
```

```
  public static void sort(int[] a){
        if (a == null || a.length == 0)
            return;
//        helperShuffle(a); // to be added to the enhanced Quicksort class
to improve overall performance
        partition(a, 0, a.length-1);
    }
```

Q2. Implement Enhanced Quick Sort.

I.

```
else if(a.length <= 10) // 1) adding a cutoff for small sub-arrays,
improves overall performance.
    insertionSort(a);
```

II.

```
helperShuffle(a); // 2) randomly shuffling the input first to improve
performance and protect against the worst case performance
```

III.

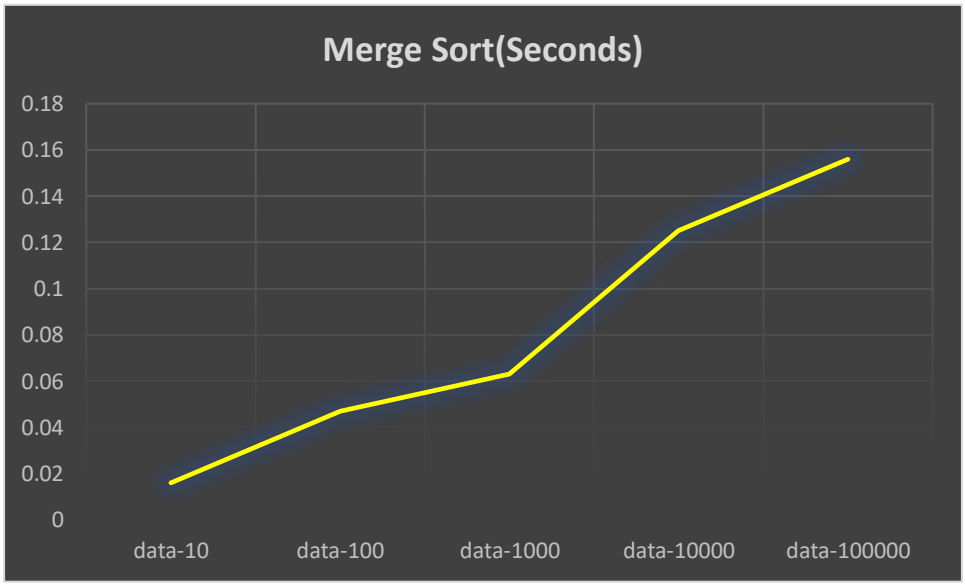
```
int pivot = a[low+(high-low)/2]; // 3) partition where value is near the
middle - median
```

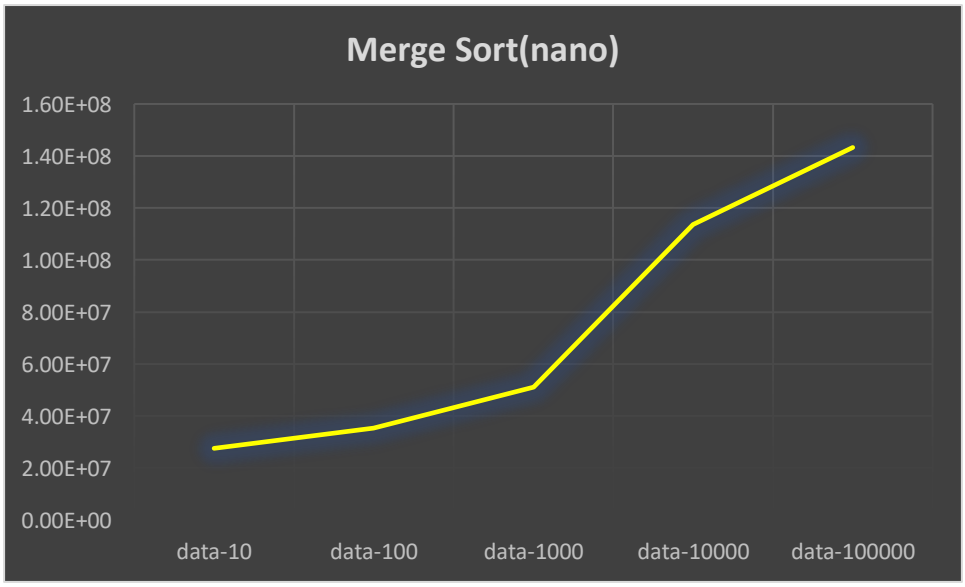
Q3. Compare Merge Quick and QuickEnhanced to eachother.

Each sort is run 5 times with inputs varying from 10-100000.

Algorithm	Inputs	Time(Seconds)	Time(nano)
Merge Sort	data-10	0.016	2.76E+07
	data-100	0.047	3.54E+07
	data-1000	0.063	5.12E+07
	data-10000	0.125	1.14E+08
	data-100000	0.156	1.43E+08

Runs in O(nlog(n)) time. Efficent algorithm for higher input sorts.

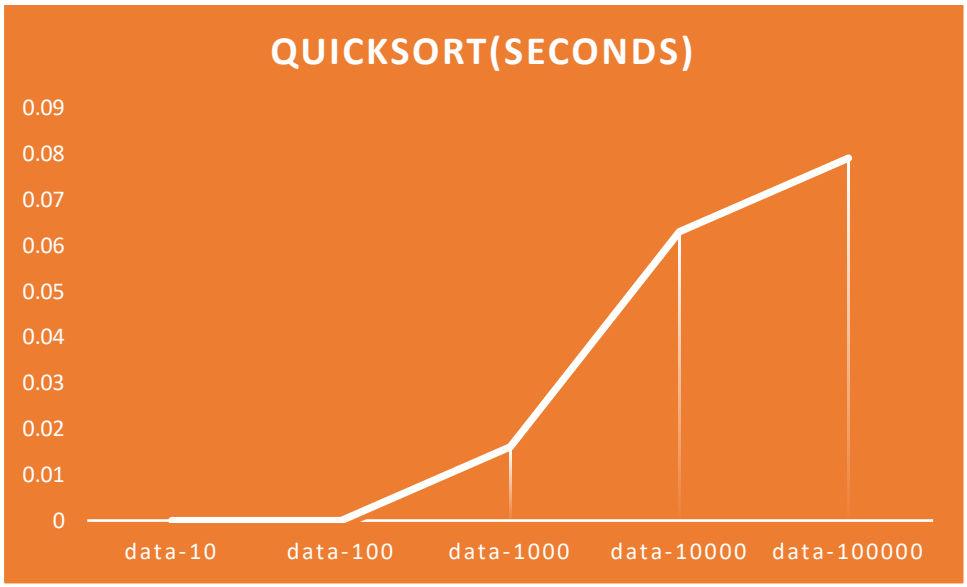




QuickSort

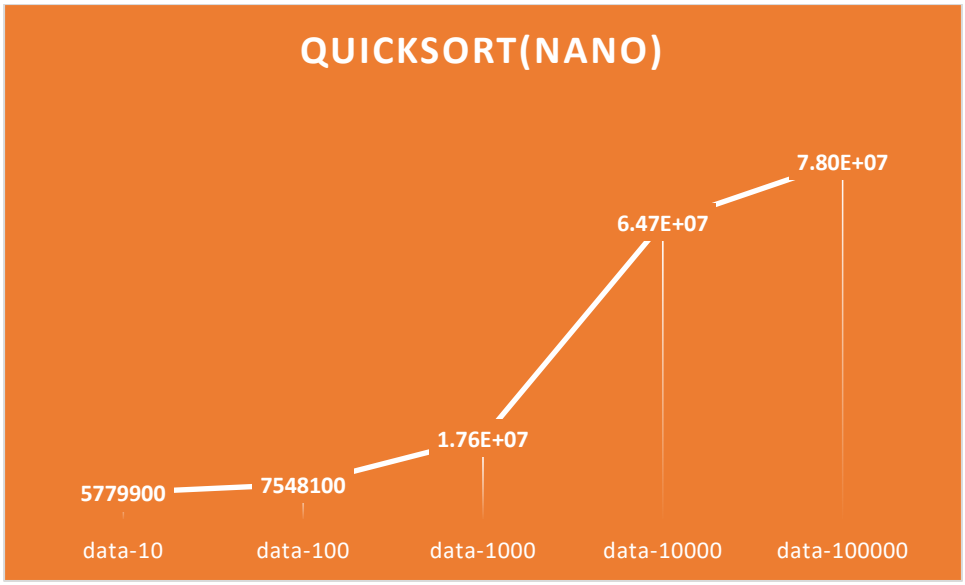
- **Very fast sorting for arrays Uses Divide and Conquer Recursion**
- **Best: O(nlogn)**
- **Average: 2NlogN**
- **Worst: 1/2N^2 – extremly unlikely case.**

Algorithm	Inputs	Time(Seconds)	Time(nano)
QuickSort	data-10	0	5779900
	data-100	0	7548100
	data-1000	0.016	1.76E+07
	data-10000	0.063	6.47E+07
	data-100000	0.079	7.80E+07



SPACE Complexity

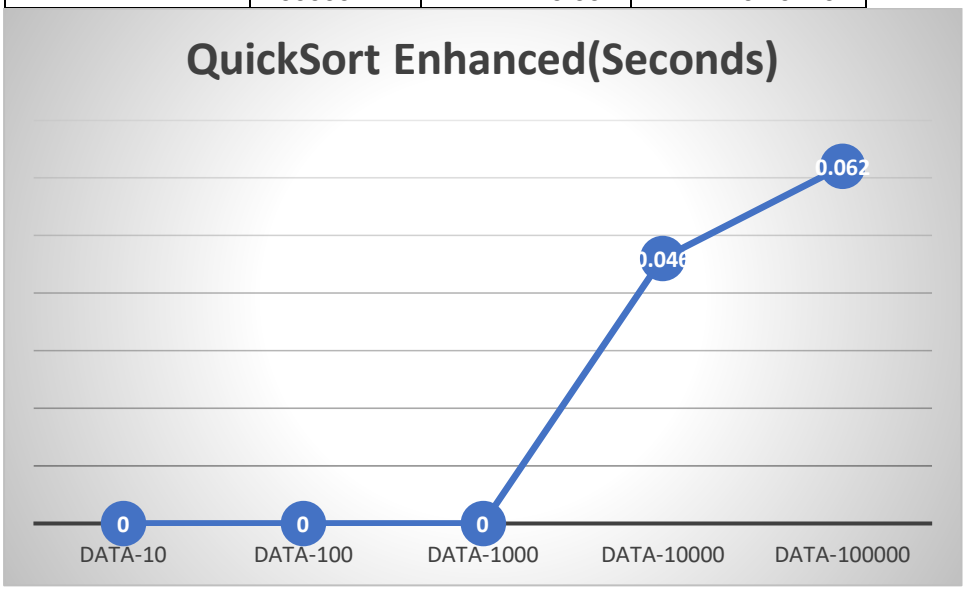
: O(log(n))

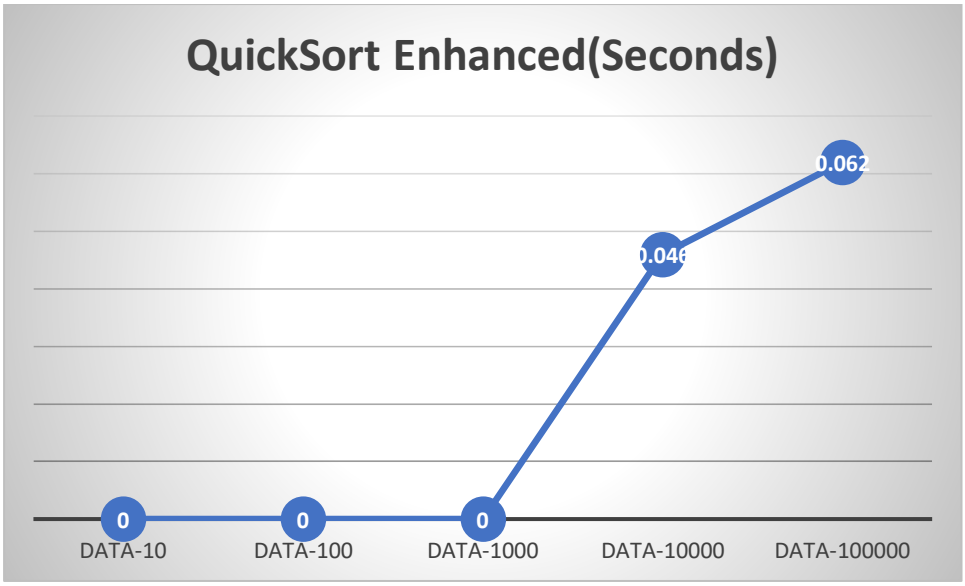


QuickSortEnhanced

- Improved version
- If input is < 10 insertion sort is run which improves timing by 10-20% as we can see below

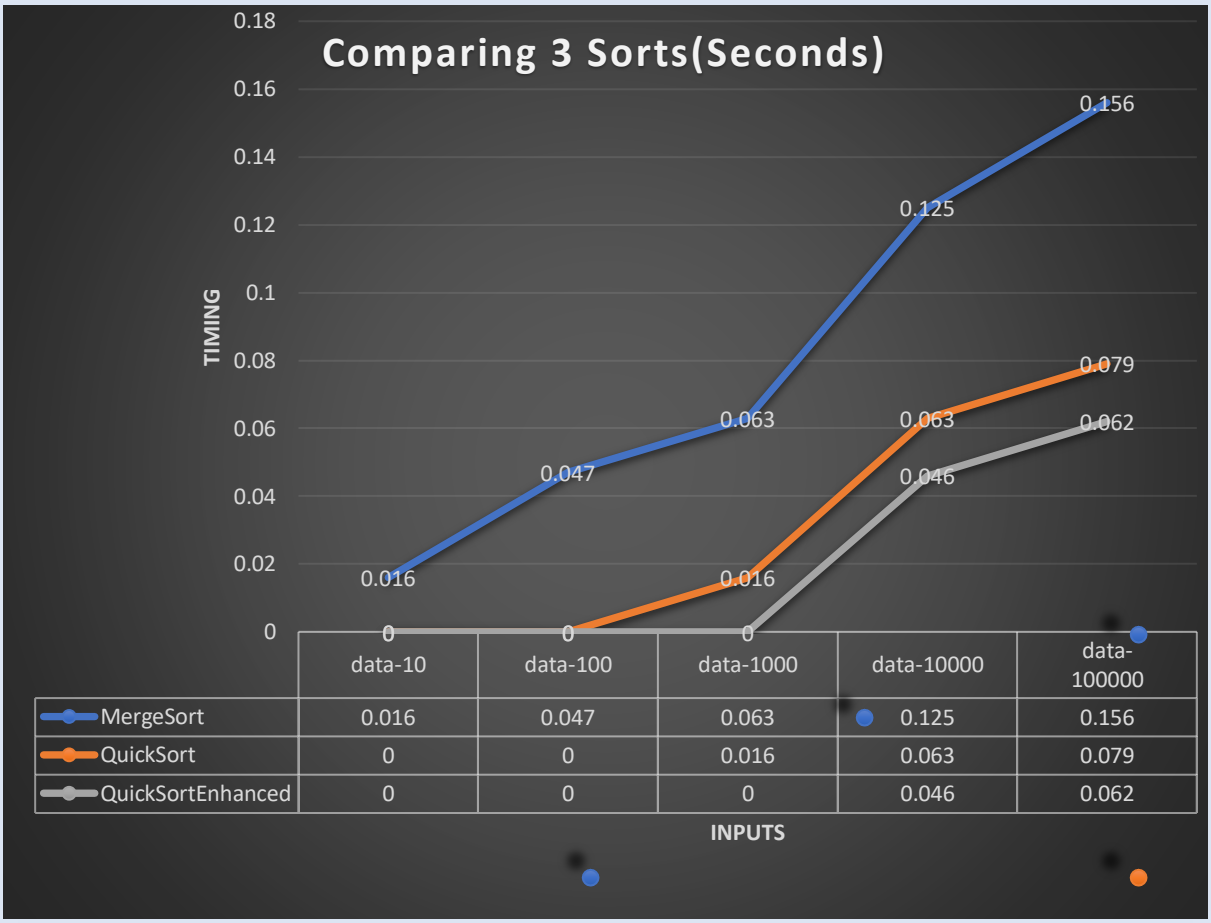
Algorithm	Inputs	Time(Seconds)	Time(nano)
QuickSort Enhanced	data-10	0	1668900
	data-100	0	2400100
	data-1000	0	6084100
	data-10000	0.046	4.54E+07
	data-100000	0.062	6.20E+07





Evidently this is the quickest sorting algorithm we have analysied to date when dealing with these inputs.

Finally Graphs comparing all 3 sorting algorithms together



This Graph clearly displays the improvement which occur when implementing quickSortEnhanced (grey line) opposed to using the initial Quicksort.

QuickSort is extremly efficent when dealing with small input arrays and significantly quicker when merged with InsertionSort.

Conclusion: QuickSortEnhanced is great when dealing with small input arrays and so far the quickest algorithm we have when dealing with large input files aswell.