Practical 6: Advanced Sorting Algorithms

What am I doing today?
Today's practical focuses on:

1. Implement Quicksort from pseudo-code
2. Develop a separate Enhanced QuickSort algorithm
3. Assess the performance difference between QuickSort and Enhanced QuickSort

Instructions
Try all the questions. Ask for help from the demonstrators if you get stuck.

***Grading: Remember** if you complete the practical, add the code to your GitHub repo which needs to be submitted at the end of the course **for an extra 5%**

Algorithmic Development

Part 1

Let's start by implementing a version of the Quick Sort algorithm (using the pseudo-code below) that sort values in ascending order.

***Don't forget to add this function to your class of sorts that you created last week.**

First implement Quick Sort:

Remember Quick Sort is an algorithm that takes a divide-and-conquer approach. The steps are:

1. Pick an element, called a pivot, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.To implement QuickSort you will need to create two functions:

1. **QuickSort**

```
/* low  --> Starting index,  high  --> Ending index
*/
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

2. **Partition**

```
partition (arr[], low, high)
{
   // pivot (Element to be placed at right position)
```

```
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Part 2

Write a second version of QuickSort (you can call it enhancedQuickSort) that implements the two improvements that we covered in the lecture:

1) Similar to your mergesort implementation last week, add a cutoff for small subarrays (e.g., <10) and use the insertion sort algorithm you wrote before to handle them. We can improve most recursive algorithms by handling small cases differently.

   Pseudo-code:

   ```
   if (hi <= lo + CUTOFF) {
       insertionSort(array, lo, hi);
           return;
       }
   ```

2) As we saw in the lecture, random shuffling the input array first improves performance and protects against the worse case performance. Add a shuffle function that takes the input array and shuffles the elements. You can use one of the helper shuffle algorithms in the rep.
3) As we saw in the lecture choosing a partition where the value is near the middle or exactly the middle of the elements in the arrays values means our sort will perform better. In quicksort with median-of-three partitioning the pivot item is selected as the median between the first element, the last element, and the middle element (decided using integer division of n/2). In the cases of already sorted lists this should take the middle element as the pivot thereby reducing the inefficency found in normal quicksort.

Look at the first, middle and last elements of the array, and choose the median of those three elements as the pivot.

E.g., int median = medianOf3(a, lo, lo + (hi - lo)/2, hi);

Solutions:

Sample implementation of median-of-three partitioning:
https://www.java-tips.org/java-se-tips-100019/24-java-lang/1896-quick-sort-implementation-with-median-of-three-partitioning-and-cutoff-for-small-arrays.html

An enhanced (or proper) version of median-of-three partition selection involves *sorting* the three items you select, not just use the median as the pivot. This doesn't affect what's chosen as the pivot in the current iteration, but can/will affect what's used as the pivot in the next recursive call, which helps to limit the bad behavior for a few initial orderings (one that turns out to be particularly bad in many cases is an array that's sorted, except for having the smallest element at the high end of the array (or largest element at the low end).

E.g., swap(a, lo, median);

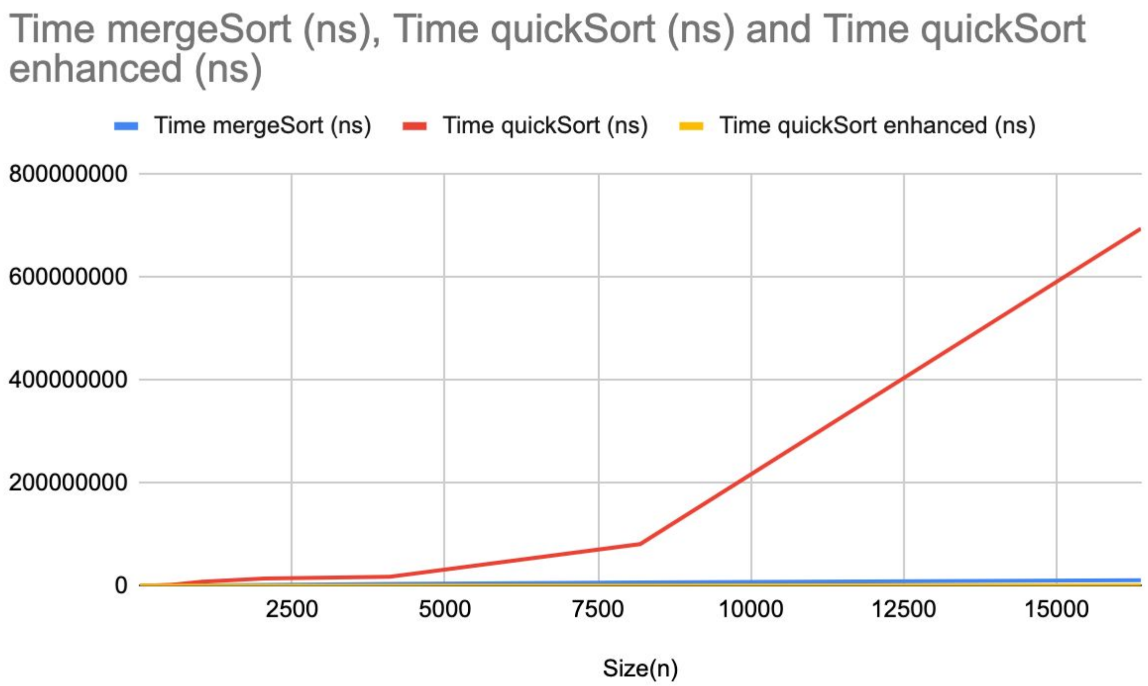
Part 3

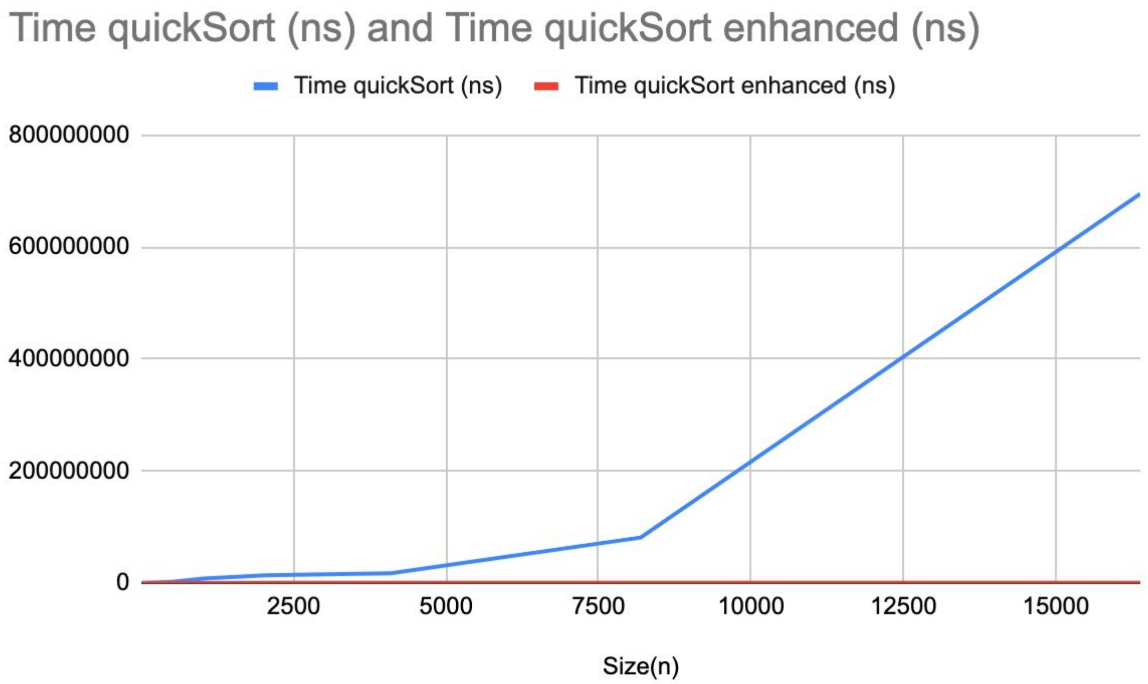
Compare the performance of MergeSort, QuickSort and QuickSortEnhanced on a range of inputs (N= 10, 1000, 10000, 100000 etc.) and graph the results of your experiments.

Experiments:

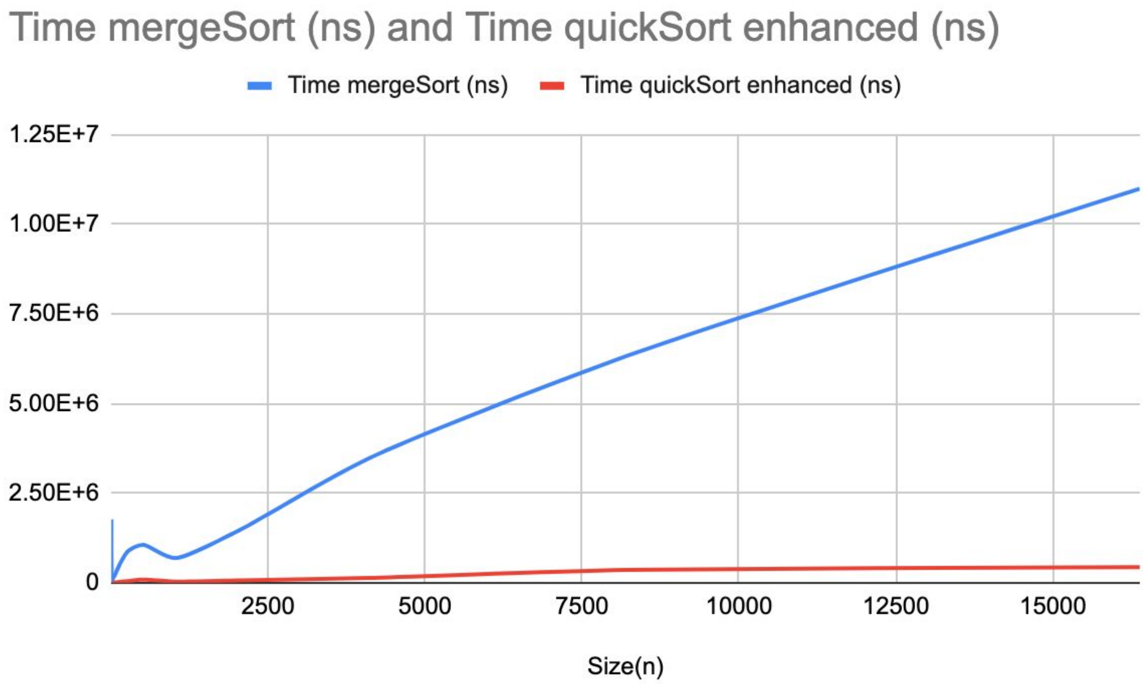
Size(n)	Time mergeSort (ns)	Time quickSort (ns)	Time quickSort enhanced (ns)
4	1771656	430991	10024
8	30927	9284	1973
16	67944	25459	4620
32	128509	88855	5993
64	233070	310833	10858
128	481019	578215	23184
256	861224	520641	46543
512	1063871	1852718	89962
1024	688187	8039054	34789
2048	1471415	13926680	66183
4096	3464517	17421372	134718
8192	6313171	81057338	358851
16384	10995106	694929215	440785

Graphs:





Most possibly the configurations of the input are benefiting the enhanced quickSort hence the difference on asymptotic behaviour.



We observe in this chart the difference between quickSort and mergeSort, where both seem to have a log asymptotic behaviour. Maybe with larger sizes of the input, we could have seen this behaviour better but because of limitations with computational power in the device performing the experiments we could not go further.