Practical 6: Advanced Sorting Algorithms II

Name: Rajit Banerjee
Student Number: 18202817

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

Algorithmic Development

Part 1

Let's start by implementing a version of the Quick Sort that sort values in ascending order.

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 worst-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 inefficiency 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);`

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.

QuickSortTest.java output:
Timing analysis

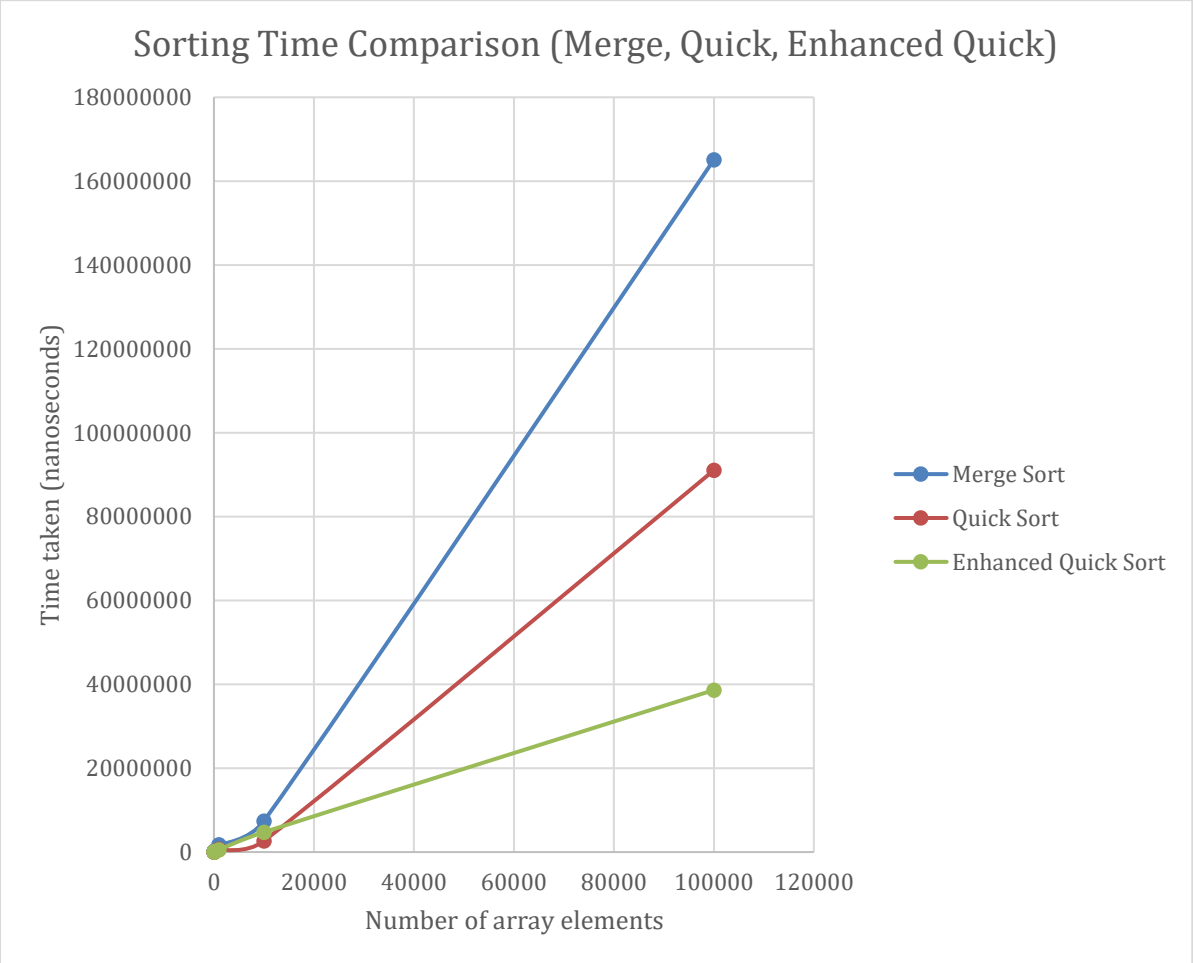
```
-Sorting Analysis-
1. Run timing analysis.
2. See sorted arrays (only small sizes).
Choose 1 or 2: 1
------------------------------------------------------------
-MERGE_SORT-
Time taken for array of size 10 = 127000 nanoseconds
Time taken for array of size 15 = 41000 nanoseconds
Time taken for array of size 100 = 275700 nanoseconds
Time taken for array of size 1000 = 1766800 nanoseconds
Time taken for array of size 10000 = 7413700 nanoseconds
Time taken for array of size 100000 = 165088600 nanoseconds
------------------------------------------------------------
-QUICK_SORT-
Time taken for array of size 10 = 25200 nanoseconds
Time taken for array of size 15 = 9400 nanoseconds
Time taken for array of size 100 = 46200 nanoseconds
Time taken for array of size 1000 = 642300 nanoseconds
Time taken for array of size 10000 = 2667500 nanoseconds
Time taken for array of size 100000 = 91046700 nanoseconds
------------------------------------------------------------
-ENHANCED_QUICK_SORT-
Time taken for array of size 10 = 182800 nanoseconds
Time taken for array of size 15 = 11400 nanoseconds
Time taken for array of size 100 = 51200 nanoseconds
Time taken for array of size 1000 = 558400 nanoseconds
Time taken for array of size 10000 = 4740800 nanoseconds
Time taken for array of size 100000 = 38632900 nanoseconds
```

Visualising the sorts:

```
-Sorting Analysis-
1. Run timing analysis.
2. See sorted arrays (only small sizes).
Choose 1 or 2: 2

Array size: 10
BEFORE MERGE_SORT:  [7, 5, 3, 8, 8, 5, 2, 6, 3, 8]
AFTER MERGE_SORT:   [2, 3, 3, 5, 5, 6, 7, 8, 8, 8]

Array size: 15
BEFORE MERGE_SORT:  [4, 6, 7, 6, 12, 7, 6, 2, 10, 8, 7, 10, 6, 13, 0]
AFTER MERGE_SORT:   [0, 2, 4, 6, 6, 6, 6, 7, 7, 7, 8, 10, 10, 12, 13]

Array size: 10
BEFORE QUICK_SORT:  [7, 5, 3, 8, 8, 5, 2, 6, 3, 8]
AFTER QUICK_SORT:   [2, 3, 3, 5, 5, 6, 7, 8, 8, 8]

Array size: 15
BEFORE QUICK_SORT:  [4, 6, 7, 6, 12, 7, 6, 2, 10, 8, 7, 10, 6, 13, 0]
AFTER QUICK_SORT:   [0, 2, 4, 6, 6, 6, 6, 7, 7, 7, 8, 10, 10, 12, 13]

Array size: 10
BEFORE ENHANCED_QUICK_SORT: [7, 5, 3, 8, 8, 5, 2, 6, 3, 8]
AFTER ENHANCED_QUICK_SORT:  [2, 3, 3, 5, 5, 6, 7, 8, 8, 8]

Array size: 15
BEFORE ENHANCED_QUICK_SORT: [4, 6, 7, 6, 12, 7, 6, 2, 10, 8, 7, 10, 6, 13, 0]
AFTER ENHANCED_QUICK_SORT:  [0, 2, 4, 6, 6, 6, 6, 7, 7, 7, 8, 10, 10, 12, 13]
```

Graphing the results:

Array Size	Time (in nanoseconds)		
	Merge Sort	Quick Sort	Enhanced Quick Sort
10	127000	25200	182800
15	41000	9400	11400
100	275700	46200	51200
1000	1766800	642300	558400
10000	7413700	2667500	4740800
100000	165088600	91046700	38632900



Evidently, merge sort is nowhere as fast as quick sort. Moreover, quick sort can be further enhanced with some optimisations to deal with small subarrays and using median of three partitioning.