Practical 6: Advanced Sorting Algorithms

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

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 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:

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:

```
-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 5 = 123900 nanosec
Time taken for array of size 10 = 25900 nanosec
Time taken for array of size 100 = 139100 nanosec
Time taken for array of size 1000 = 1148400 nanosec
Time taken for array of size 10000 = 4371400 nanosec
Time taken for array of size 100000 = 24895800 nanosec
Time taken for array of size 200000 = 37046500 nanosec
-----------------------------------------------------------
-QUICK SORT-
Time taken for array of size 5 = 23000 nanosec
Time taken for array of size 10 = 6600 nanosec
Time taken for array of size 100 = 38300 nanosec
Time taken for array of size 1000 = 322700 nanosec
Time taken for array of size 10000 = 1015200 nanosec
Time taken for array of size 100000 = 9194800 nanosec
Time taken for array of size 200000 = 16560900 nanosec
-----------------------------------------------------------
-QUICK_ENHANCED SORT-
Time taken for array of size 5 = 20000 nanosec
Time taken for array of size 10 = 5300 nanosec
Time taken for array of size 100 = 17700 nanosec
Time taken for array of size 1000 = 122000 nanosec
Time taken for array of size 10000 = 918900 nanosec
Time taken for array of size 100000 = 6880600 nanosec
Time taken for array of size 200000 = 14275600 nanosec
```

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

ORIGINAL ARRAY (size 5):    { 1, 3, 3, 2, 3 }
AFTER MERGE SORT:       { 1, 2, 3, 3, 3 }

ORIGINAL ARRAY (size 10):   { 8, 4, 7, 1, 1, 7, 9, 5, 2, 2 }
AFTER MERGE SORT:       { 1, 1, 2, 2, 4, 5, 7, 7, 8, 9 }

ORIGINAL ARRAY (size 5):    { 1, 3, 3, 2, 3 }
AFTER QUICK SORT:       { 1, 2, 3, 3, 3 }

ORIGINAL ARRAY (size 10):   { 8, 4, 7, 1, 1, 7, 9, 5, 2, 2 }
AFTER QUICK SORT:       { 1, 1, 2, 2, 4, 5, 7, 7, 8, 9 }

ORIGINAL ARRAY (size 5):    { 1, 3, 3, 2, 3 }
AFTER QUICK_ENHANCED SORT:      { 1, 2, 3, 3, 3 }

ORIGINAL ARRAY (size 10):   { 8, 4, 7, 1, 1, 7, 9, 5, 2, 2 }
AFTER QUICK_ENHANCED SORT:      { 1, 1, 2, 2, 4, 5, 7, 7, 8, 9 }
```