Practical 5: Advanced Sorting Algorithms

Name: Rajit Banerjee
Student Number: 18202817

What am I doing today?

Today's practical focuses on:

1. Implementing MergeSort from pseudo-code and comparing MergeSort to Insertion Sort for increasing input sizes
2. Implementing enhanced MergeSort
3. Compare the performance between 3 sorting algorithms

Quick Questions

1. MergeSort guarantees to sort an array in _____ time, regardless of the input:
   A. Linear time
   B. Quadratic time
   C. Linearithmic time
   D. Logarithmic time

2. The main disadvantage of MergeSort is:
   A. It is difficult to implement
   B. It uses extra space in proportion to the size of the input
   C. It is an unstable sort
   D. None of the above

3. Merge sort makes use of which common algorithm strategy?
   A. Dynamic Programming
   B. Branch-and-bound
   C. Greedy approach
   D. Divide and conquer

4. Which sorting algorithm will take the least time when all elements of the input array are identical?
   A. Insertion Sort
   B. MergeSort
   C. Selection Sort
   D. Bogo Sort

5. Which sorting algorithm should you use when the order of input is not known?
   A. MergeSort
   B. Insertion sort
   C. Selection sort
   D. Shell sort

Algorithmic Development

Part 1

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

First implement MergeSort:

```
function MergeSort (int[] a){
N = array.length;
//base case
if (n == 1){
return array;
}
//create left and right sub-arrays
left = MergeSort(left);
right = MergeSort(right);

mergeArray = merge(left, right);

return mergedArray;
}
```

Second implement the recursive merge:

```
function merge (int[] a, int[] b){
//repeat while both arrays have elements in them
while (a.notEmpty() && b.notEmpty()){

//if element in 1st array is <= 1st element in 2nd array
if (a.firstElement <= b.firstElement){
S.insertLast(a.removeFirst());
} else if (b.firstElement <= a.firstElement){
S.insertLast(b.removeFirst());
}

//when while loop ends
If (a.notEmpty()){
//add remaining elements in a to S
} else if (b.notEmpty()){
//add remaining elements in b to S
}
return S;
```

Part 2

Write a second version of MergeSort that implements the two improvements to MergeSort that we covered in the lecture:

1) add a cut-off for small subarrays and use insertion sort (written last time) to handle them. We can improve most recursive algorithms by handling small cases differently.

Pseudo-code:

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

2) test whether the array is already in order. We can reduce the running time to be linear for arrays that are already in order by adding a test to skip call to merge() if a[mid] is less than or equal to a[mid+1]. **In other words, if the last element in the first sorted array is less than or equal to the first element in the second sorted array then you can just add the entire second array in without the need for comparisons.** With this change, we still do all the recursive calls, but the running time for any sorted subarray is linear.

Part 3

Compare the performance of Insertion Sort, MergeSort and MergeSortEnhanced on a range of inputs (N= 10, 1000, 10000, 100000 etc.).

Solution note: Switching to insertion sort for small subarrays will improve the running time of a typical MergeSort implementation by 10 to 15 percent.

MergeSortTest.java output:
Timing analysis

```
-Sorting Analysis-
1. Run timing analysis.
2. See sorted arrays (only small sizes).
Choose 1 or 2: 1
------------------------------------------------------------
-ENHANCED_MERGE_SORT-
Time taken for array of size 10 = 100100 nanoseconds
Time taken for array of size 15 = 13900 nanoseconds
Time taken for array of size 100 = 112100 nanoseconds
Time taken for array of size 1000 = 758100 nanoseconds
Time taken for array of size 10000 = 3868800 nanoseconds
Time taken for array of size 100000 = 95411100 nanoseconds
------------------------------------------------------------
-MERGE_SORT-
Time taken for array of size 10 = 42200 nanoseconds
Time taken for array of size 15 = 54400 nanoseconds
Time taken for array of size 100 = 115200 nanoseconds
Time taken for array of size 1000 = 1425500 nanoseconds
Time taken for array of size 10000 = 13609100 nanoseconds
Time taken for array of size 100000 = 182815900 nanoseconds
------------------------------------------------------------
-INSERTION_SORT-
Time taken for array of size 10 = 25500 nanoseconds
Time taken for array of size 15 = 17800 nanoseconds
Time taken for array of size 100 = 21000 nanoseconds
Time taken for array of size 1000 = 1281600 nanoseconds
Time taken for array of size 10000 = 137560300 nanoseconds
Time taken for array of size 100000 = 18319241800 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 ENHANCED_MERGE_SORT: [3, 9, 4, 7, 8, 6, 8, 1, 6, 6]
AFTER ENHANCED_MERGE_SORT:  [1, 3, 4, 6, 6, 6, 7, 8, 8, 9]

Array size: 15
BEFORE ENHANCED_MERGE_SORT: [5, 13, 7, 13, 11, 4, 1, 1, 7, 7, 4, 8, 2, 4, 9]
AFTER ENHANCED_MERGE_SORT:  [1, 1, 2, 4, 4, 4, 5, 7, 7, 7, 8, 9, 11, 13, 13]

Array size: 10
BEFORE MERGE_SORT:  [3, 9, 4, 7, 8, 6, 8, 1, 6, 6]
AFTER MERGE_SORT:   [1, 3, 4, 6, 6, 6, 7, 8, 8, 9]

Array size: 15
BEFORE MERGE_SORT:  [5, 13, 7, 13, 11, 4, 1, 1, 7, 7, 4, 8, 2, 4, 9]
AFTER MERGE_SORT:   [1, 1, 2, 4, 4, 4, 5, 7, 7, 7, 8, 9, 11, 13, 13]

Array size: 10
BEFORE INSERTION_SORT:  [3, 9, 4, 7, 8, 6, 8, 1, 6, 6]
AFTER INSERTION_SORT:   [1, 3, 4, 6, 6, 6, 7, 8, 8, 9]

Array size: 15
BEFORE INSERTION_SORT:  [5, 13, 7, 13, 11, 4, 1, 1, 7, 7, 4, 8, 2, 4, 9]
AFTER INSERTION_SORT:   [1, 1, 2, 4, 4, 4, 5, 7, 7, 7, 8, 9, 11, 13, 13]
```

Graphing the results:

Array Size	Time (in nanoseconds)		
	Insertion Sort	Merge Sort	Enhanced Merge Sort
10	25500	42200	100100
15	17800	54400	13900
100	21000	115200	112100
1000	1281600	1425500	758100
10000	137560300	13609100	3868800
100000	18319241800	182815900	95411100