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.

***please add this function to your class of sorts that you created last week.**

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;
    }
```

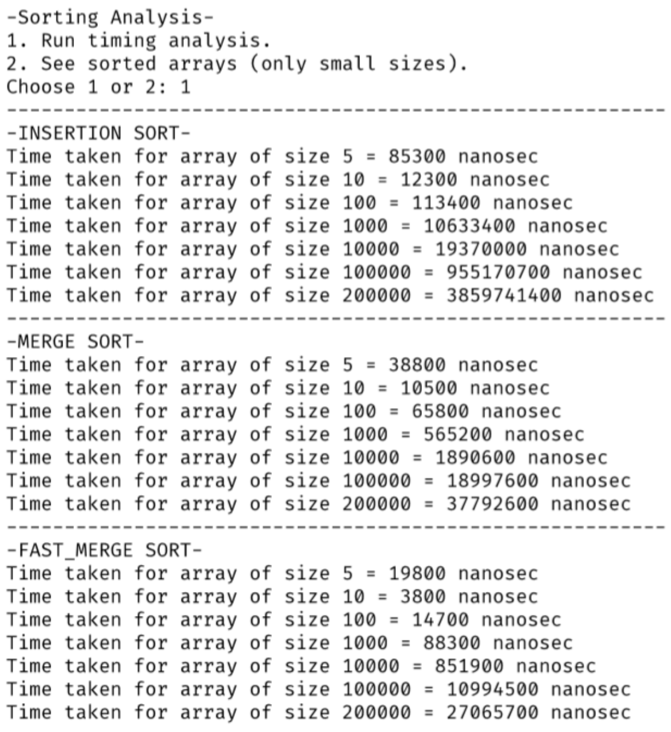
2) test whether the array is already in order. We can reduce the running time to be linear for arrays that are already in order by adding a test to skip call to merge() if a[mid] is less than or equal to a[mid+1]. **In other words, if the last element in the first sorted array is less than or equal to 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

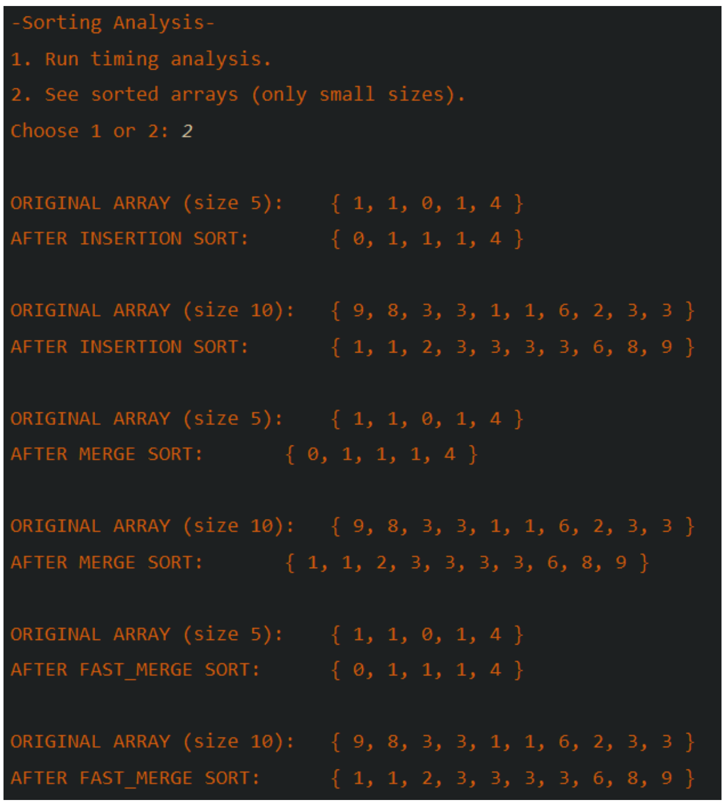
Compare the performance of Insertion Sort, MergeSort and MergeSortEnhanced on a range of inputs (N= 10, 1000, 10000, 100000 etc.).

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:



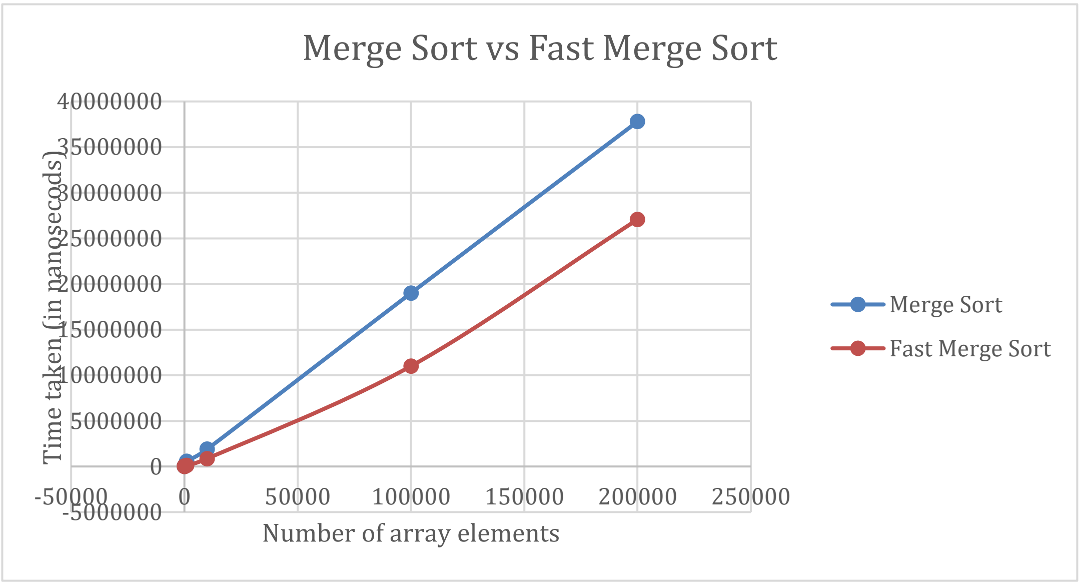
```
-Sorting Analysis-
1. Run timing analysis.
2. See sorted arrays (only small sizes).
Choose 1 or 2: 1
---------------------------------------------------------
-INSERTION SORT-
Time taken for array of size 5 = 85300 nanosec
Time taken for array of size 10 = 12300 nanosec
Time taken for array of size 100 = 113400 nanosec
Time taken for array of size 1000 = 10633400 nanosec
Time taken for array of size 10000 = 19370000 nanosec
Time taken for array of size 100000 = 955170700 nanosec
Time taken for array of size 200000 = 3859741400 nanosec
---------------------------------------------------------
-MERGE SORT-
Time taken for array of size 5 = 38800 nanosec
Time taken for array of size 10 = 10500 nanosec
Time taken for array of size 100 = 65800 nanosec
Time taken for array of size 1000 = 565200 nanosec
Time taken for array of size 10000 = 1890600 nanosec
Time taken for array of size 100000 = 18997600 nanosec
Time taken for array of size 200000 = 37792600 nanosec
---------------------------------------------------------
-FAST_MERGE SORT-
Time taken for array of size 5 = 19800 nanosec
Time taken for array of size 10 = 3800 nanosec
Time taken for array of size 100 = 14700 nanosec
Time taken for array of size 1000 = 88300 nanosec
Time taken for array of size 10000 = 851900 nanosec
Time taken for array of size 100000 = 10994500 nanosec
Time taken for array of size 200000 = 27065700 nanosec
```

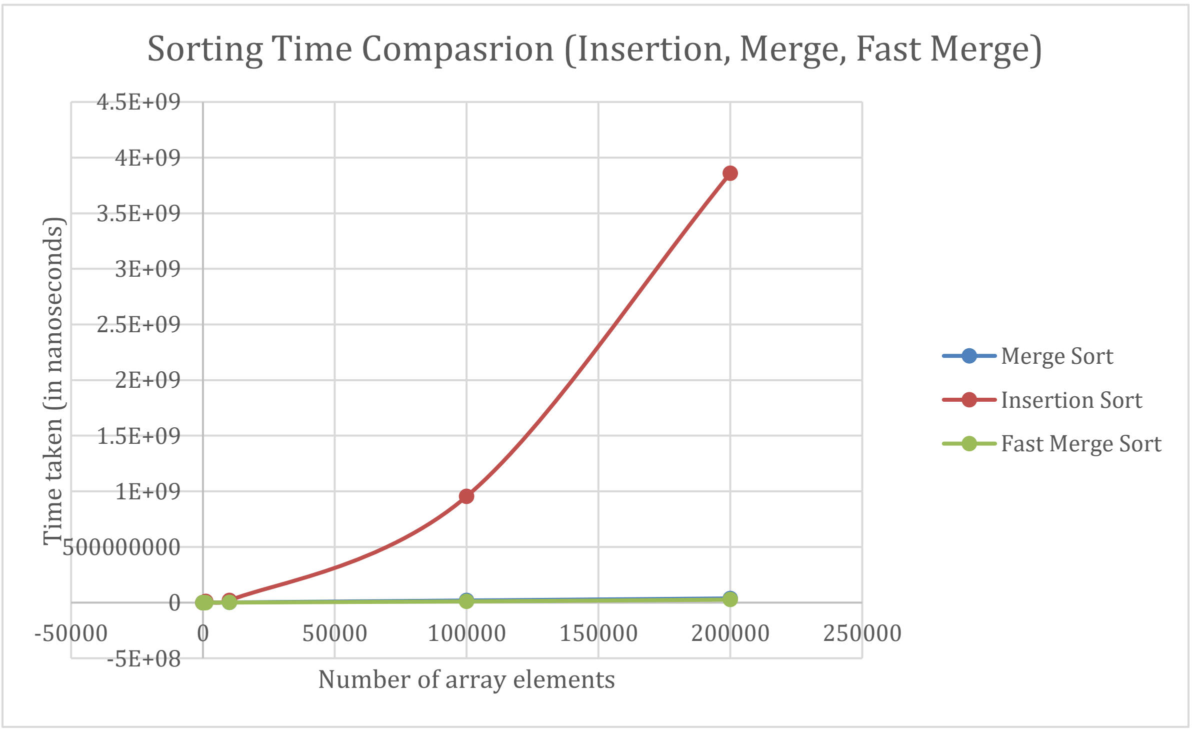


```
-Sorting Analysis-
1. Run timing analysis.
2. See sorted arrays (only small sizes).
Choose 1 or 2: 2

ORIGINAL ARRAY (size 5):    { 1, 1, 0, 1, 4 }
AFTER INSERTION SORT:       { 0, 1, 1, 1, 4 }

ORIGINAL ARRAY (size 10):   { 9, 8, 3, 3, 1, 1, 6, 2, 3, 3 }
AFTER INSERTION SORT:       { 1, 1, 2, 3, 3, 3, 3, 6, 8, 9 }

ORIGINAL ARRAY (size 5):    { 1, 1, 0, 1, 4 }
AFTER MERGE SORT:       { 0, 1, 1, 1, 4 }

ORIGINAL ARRAY (size 10):   { 9, 8, 3, 3, 1, 1, 6, 2, 3, 3 }
AFTER MERGE SORT:       { 1, 1, 2, 3, 3, 3, 3, 6, 8, 9 }

ORIGINAL ARRAY (size 5):    { 1, 1, 0, 1, 4 }
AFTER FAST_MERGE SORT:      { 0, 1, 1, 1, 4 }

ORIGINAL ARRAY (size 10):   { 9, 8, 3, 3, 1, 1, 6, 2, 3, 3 }
AFTER FAST_MERGE SORT:      { 1, 1, 2, 3, 3, 3, 3, 6, 8, 9 }
```

Graphing the results:

Array Size	Time (in nanoseconds)		
	Insertion Sort	Merge Sort	Fast Merge Sort
5	85300	38800	19800
10	12300	10500	3800
100	113400	65800	14700
1000	10633400	565200	88300
10000	19370000	1890600	851900
100000	955170700	18997600	10994500
200000	3859741400	37792600	27065700


Merge Sort vs Fast Merge Sort
Time taken (in nanoseconds)
Number of array elements
Merge Sort
Fast Merge Sort


Sorting Time Compasrion (Insertion, Merge, Fast Merge)
4.5E+09
4E+09
3.5E+09
3E+09
2.5E+09
2E+09
1.5E+09
1E+09
-5E+08
Time taken (in nanoseconds)
Number of array elements
Merge Sort
Insertion Sort
Fast Merge Sort