**PROJECT 1**

**Comparison-based Sorting Algorithms**

**JINRAJ JAIN**

**ANKIT PANDITA**

**Introduction**

**Sort Algorithms:**

A sorting algorithm is an algorithm made up of a series of instructions that takes an array as input, performs specified operations on the array, sometimes called a list, and outputs a sorted array. A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure

**Insertion Sort:**

Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e. the position to which it belongs in a sorted array. It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array. If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead.

**Time Complexity:**

In worst case, each element is compared with all the other elements in the sorted array. For 𝑁 elements, there will be 𝑁2 comparisons. Therefore, the time complexity is Ο(𝑁2)

Code:

**package** edu.uncc.cci.algods;  
  
*/\*  
Project 01  
Authors: Ankit Pandita, Jinraj Jain  
\*/***class** InsertionSort {  
 **void** performInsertionSort(**int**[] intArray) {  
 **int** size = intArray.**length**;  
 **for** (**int** j = 1; j <= size-1; j++) {  
 **int** key = intArray[j];  
 **int** i = j - 1;  
 **while** ((i >= 0) && (intArray[i] > key)) {  
 intArray[i + 1] = intArray[i];  
 i--;  
 }  
 intArray[i + 1] = key;  
 }  
 }  
}

**Merge Sort:**

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

Idea:

• Divide the unsorted list into 𝑁 sublists, each 1 containing element.

• Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. 𝑁 will now convert into 𝑁/2 lists of size 2.

• Repeat the process till a single sorted list of obtained.

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.

**Time Complexity:**

The list of size 𝑁 is divided into a max of log(𝑁) parts, and the merging of all sublists into a single list takes Ο(𝑁) time, the worst case run time of this algorithm

is Ο(𝑁𝐿𝑜𝑔𝑁).

**class** MergeSort {  
 **private void** merge(**int**[] intArray, **int** startIndex, **int** middleIndex, **int** lastIndex) {  
 **int** l = middleIndex - startIndex + 1;  
 **int** r = lastIndex - middleIndex;  
 **int** i, j;  
 **int**[] LeftArray = **new int**[l];  
 **int**[] RightArray = **new int**[r];  
 **for** (i = 0; i < l; i++) {  
 LeftArray[i] = intArray[startIndex + i];  
 }  
 **for** (j = 0; j < r; j++) {  
 RightArray[j] = intArray[middleIndex + 1 + j];  
 }  
 i = 0;  
 j = 0;  
 **int** k = startIndex;  
 **while** (i < l && j < r) {  
 **if** (LeftArray[i] <= RightArray[j]) {  
 intArray[k] = LeftArray[i];  
 i++;  
 } **else** {  
 intArray[k] = RightArray[j];  
 j++;  
 }  
 k++;  
 }  
 **while** (i < l) {  
 intArray[k] = LeftArray[i];  
 i++;  
 k++;  
 }  
  
 **while** (j < r) {  
 intArray[k] = RightArray[j];  
 j++;  
 k++;  
 }  
 }  
  
 **private void** performMergeSort(**int**[] intArray, **int** startIndex, **int** lastIndex) {  
 **if** (startIndex < lastIndex) {  
 **int** middleIndex = (startIndex + lastIndex) / 2;  
 performMergeSort(intArray, startIndex, middleIndex);  
 performMergeSort(intArray, middleIndex + 1, lastIndex);  
 merge(intArray, startIndex, middleIndex, lastIndex);  
 }  
 }  
  
 **void** sort(**int**[] intArray) {  
 performMergeSort(intArray, 0, intArray.**length** - 1);  
 }  
}

**Heapsort:**

Heaps can be used in sorting an array. In max-heaps, maximum element will always be at the root. Heap Sort uses this property of heap to sort the array. Consider an array 𝐴𝑟𝑟 which is to be sorted using Heap Sort.

• Initially build a max heap of elements in 𝐴𝑟𝑟.

• The root element, that is 𝐴𝑟𝑟 [1] , will contain maximum element of 𝐴𝑟𝑟. After that, swap this element with the last element of 𝐴𝑟𝑟 and **heapify** the max heap excluding the last element which is already in its correct position and then decrease the length of heap by one.

• Repeat the step 2, until all the elements are in their correct position.

**Time Complexity:**

Heapify has complexity Ο(𝐿𝑜𝑔𝑁) , build\_maxheap has complexity Ο(𝑁) and we run max\_heapify 𝑁 − 1 times in heap\_sort function, therefore complexity of

heap\_sort function is Ο(𝑁𝐿𝑜𝑔𝑁).

**package** edu.uncc.cci.algods;  
  
*/\*  
Project 01  
Authors: Ankit Pandita, Jinraj Jain  
\*/***class** HeapSort {  
 **private void** heapify(**int**[] intArray, **int** size, **int** i) {  
 *// Initialize largest as root* **int** largest = i;  
 **int** left = 2 \* i + 1;  
 **int** right = 2 \* i + 2;  
 **int** temp;  
 *// If left child is larger than root* **if** (left < size && intArray[left] > intArray[largest]) {  
 largest = left;  
 }  
 *// If right child is larger than largest so far* **if** (right < size && intArray[right] > intArray[largest]) {  
 largest = right;  
 }  
 *// If largest is not root* **if** (largest != i) {  
 temp = intArray[i];  
 intArray[i] = intArray[largest];  
 intArray[largest] = temp;  
 *// Recursively heapify the affected sub-tree* heapify(intArray, size, largest);  
 }  
 }  
  
 **void** performHeapSort(**int**[] intArray) {  
 **int** size = intArray.**length**;  
 **int** i;  
 **int** temp;  
 *// Build heap (rearrange array)* **for** (i = size / 2 - 1; i >= 0; i--) {  
 heapify(intArray, size, i);  
 }  
 *// One by one extract an element from heap* **for** (i = size - 1; i >= 0; i--) {  
 *// Move current root to end* temp = intArray[0];  
 intArray[0] = intArray[i];  
 intArray[i] = temp;  
 *// call max heapify on the reduced heap* heapify(intArray, i, 0);  
 }

**Quick sort:**

Another divide and conquer algorithm, but with the hard work, comparison, done in the divide stage.

Assume the problem is a Sequence S of n unordered objects and we want to return

S sorted. Using the Divide and conquer design pattern:

1. Divide: If S has two or more elements select a specific element called the pivot. (frequently the pivot is the last element.) Remove the elements from S and put them into three sequences:

o L, storing elements less than the pivot.

2. E, storing elements equal to the pivot

3. G, storing elements greater than the pivot.

4. Recur: Recursively sort sequences L and G.

5. Conquer: Put back the elements into S by first inserting L then E and finally G.

6. Note that the height of the quick-sort tree is expected to be O(logN) but can be as bad as O(N) with bad pivots. Using the last element as pivot then a sorted sequence will take O(N). The height of the quick-sort tree is depended on the choice of pivots.

**Time Complexity: Inplace Quicksort**

Quicksort has worst case complexity Ο(𝑁2) , best case complexity

O(𝑁𝐿𝑜𝑔𝑁).

**Inplace Quicksort**

**package** edu.uncc.cci.algods;  
  
*/\*  
Project 01  
Authors: Ankit Pandita, Jinraj Jain  
\*/***class** InPlaceQuickSort {  
 **private int**[] **input**;  
  
 **void** performInPlaceQuickSort(**int**[] numbers) {  
 **if** (numbers == **null** || numbers.**length** == 0) {  
 **return**;  
 }  
 **this**.**input** = numbers;  
 **int** length = numbers.**length**;  
 quickSort(0, length - 1);  
 }  
  
 */\* This method implements in-place quick sort algorithm recursively. \*/* **private void** quickSort(**int** low, **int** high) {  
 **int** i = low;  
 **int** j = high;  
 *// pivot is middle index* **int** pivot = **input**[low + (high - low) / 2];  
 *// Divide into two arrays* **while** (i <= j) {  
 **while** (**input**[i] < pivot) {  
 i++;  
 }  
 **while** (**input**[j] > pivot) {  
 j--;  
 }  
 **if** (i <= j) {  
 swap(i, j);  
 *// move index to next position on both sides* i++;  
 j--;  
 }  
 }  
 *// calls quickSort() method recursively* **if** (low < j) {  
 quickSort(low, j);  
 }  
 **if** (i < high) {  
 quickSort(i, high);  
 }  
 }  
  
 **private void** swap(**int** i, **int** j) {  
 **int** temp = **input**[i];  
 **input**[i] = **input**[j];  
 **input**[j] = temp;  
 }  
}

**Median of three Quicksort**

**package** edu.uncc.cci.algods;  
  
*/\*  
Project 01  
Authors: Ankit Pandita, Jinraj Jain  
\*/***class** MedianOfThreeQuickSort {  
  
 **void** performSort(**int**[] intArray) {  
 recQuickSort(intArray, 0, intArray.**length** - 1);  
 }  
  
 **private void** recQuickSort(**int**[] intArray, **int** left, **int** right) {  
 **int** size = right - left + 1;  
 **if** (size <= 3)  
 manualSort(intArray, left, right);  
 **else** {  
 **double** median = medianOfThree(intArray, left, right);  
 **int** partition = partitionIt(intArray, left, right, median);  
 recQuickSort(intArray, left, partition - 1);  
 recQuickSort(intArray, partition + 1, right);  
 }  
 }  
  
 **private int** medianOfThree(**int**[] intArray, **int** left, **int** right) {  
 **int** center = (left + right) / 2;  
  
 **if** (intArray[left] > intArray[center])  
 swap(intArray, left, center);  
  
 **if** (intArray[left] > intArray[right])  
 swap(intArray, left, right);  
  
 **if** (intArray[center] > intArray[right])  
 swap(intArray, center, right);  
  
 swap(intArray, center, right - 1);  
 **return** intArray[right - 1];  
 }  
  
 **private void** swap(**int**[] intArray, **int** dex1, **int** dex2) {  
 **int** temp = intArray[dex1];  
 intArray[dex1] = intArray[dex2];  
 intArray[dex2] = temp;  
 }  
  
 **private int** partitionIt(**int**[] intArray, **int** left, **int** right, **double** pivot) {  
 **int** leftPointer = left;  
 **int** rightPointer = right - 1;  
  
 **while** (**true**) {  
 **while** (intArray[++leftPointer] < pivot)  
 ;  
 **while** (intArray[--rightPointer] > pivot)  
 ;  
 **if** (leftPointer >= rightPointer)  
 **break**;  
 **else** swap(intArray, leftPointer, rightPointer);  
 }  
 swap(intArray, leftPointer, right - 1);  
 **return** leftPointer;  
 }  
  
 **private void** manualSort(**int**[] intArray, **int** left, **int** right) {  
 **int** size = right - left + 1;  
 **if** (size <= 1)  
 **return**;  
 **if** (size == 2) {  
 **if** (intArray[left] > intArray[right])  
 swap(intArray, left, right);  
 } **else** {  
 **if** (intArray[left] > intArray[right - 1])  
 swap(intArray, left, right - 1);  
 **if** (intArray[left] > intArray[right])  
 swap(intArray, left, right);  
 **if** (intArray[right - 1] > intArray[right])  
 swap(intArray, right - 1, right);  
 }  
 }  
}

**Main.java**

**package** edu.uncc.cci.algods;  
  
**import** org.math.plot.Plot2DPanel;  
  
**import** javax.swing.\*;  
**import** java.awt.\*;  
**import** java.util.\*;  
  
*/\*  
Project 01  
Authors: Ankit Pandita, Jinraj Jain  
\*/***public class** Main {  
 **public static void** main(String[] args) {  
 **int**[] arraySizes = {1000, 2000, 5000, 10000, 20000, 40000, 50000};  
 **double**[] doubleIS = **new double**[arraySizes.**length**];  
 **double**[] doubleMS = **new double**[arraySizes.**length**];  
 **double**[] doubleHS = **new double**[arraySizes.**length**];  
 **double**[] doubleIP = **new double**[arraySizes.**length**];  
 **double**[] doubleM3 = **new double**[arraySizes.**length**];  
 **int** count = 0;  
 **for** (**int** i : arraySizes) {  
 **int**[] arrNum1 = *getArray*(i);  
 **int**[] arrNum2 = *getArray*(i);  
 **int**[] arrNum3 = *getArray*(i);  
 *//sortArrayAscending(arrNum1, arrNum2, arrNum3);  
 //sortArrayDescending(arrNum1, arrNum2, arrNum3);* doubleIS[count] = *getExecutionTimeIS*(arrNum1, arrNum2, arrNum3);  
 doubleMS[count] = *getExecutionTimeMS*(arrNum1, arrNum2, arrNum3);  
 doubleHS[count] = *getExecutionTimeHS*(arrNum1, arrNum2, arrNum3);  
 doubleIP[count] = *getExecutionTimeIP*(arrNum1, arrNum2, arrNum3);  
 doubleM3[count] = *getExecutionTimeM3*(arrNum1, arrNum2, arrNum3);  
 count++;  
 }  
 **double**[] doubles = **new double**[arraySizes.**length**];  
 **for** (**int** i = 0; i < arraySizes.**length**; i++) {  
 doubles[i] = arraySizes[i];  
 }  
 *// create your PlotPanel (you can use it as a JPanel)* Plot2DPanel plot = **new** Plot2DPanel();  
 *// add a line plot to the PlotPanel* plot.addLinePlot(**"Insertion Sort"**, Color.***red***, doubles, doubleIS);  
 plot.addLinePlot(**"Merge Sort"**, Color.***green***, doubles, doubleMS);  
 plot.addLinePlot(**"Heap Sort"**, Color.***blue***, doubles, doubleHS);  
 plot.addLinePlot(**"In-Place Quick Sort"**, Color.***yellow***, doubles, doubleIP);  
 plot.addLinePlot(**"Median-of-Three Quick Sort"**, Color.***black***, doubles, doubleM3);  
 *// put the PlotPanel in a JFrame, as a JPanel* JFrame frame = **new** JFrame(**"a plot panel"**);  
 frame.setContentPane(plot);  
 frame.setVisible(**true**);  
 }  
  
 **private static int**[] getArray(**int** size) {  
 Random r = **new** Random();  
 **int**[] integers = **new int**[size];  
 **for** (**int** i = 0; i < integers.**length**; i++) {  
 integers[i] = r.nextInt();  
 }  
 **return** integers;  
 }  
  
 *//Insertion Sort* **private static long** startInsertionSort(**int**[] numberArray) {  
 **long** startTimeIS = System.*nanoTime*();  
 InsertionSort insertionSort = **new** InsertionSort();  
 insertionSort.performInsertionSort(numberArray);  
 **long** endTimeIS = System.*nanoTime*();  
 System.***out***.println(**"\n\nSorting using Insertion Sort:"**);  
 **for** (**int** i : numberArray) {  
 System.***out***.print(i + **" "**);  
 }  
 **long** execTimeIS = endTimeIS - startTimeIS;  
 System.***out***.println(**"\nTime Taken for execution using Insertion Sort = "** + execTimeIS + **" nanoseconds"**);  
 **return** execTimeIS;  
 }  
  
 *//Merge Sort* **private static long** startMergeSort(**int**[] numberArray) {  
 **long** startTimeMS = System.*nanoTime*();  
 MergeSort mergeSort = **new** MergeSort();  
 mergeSort.sort(numberArray);  
 **long** endTimeMS = System.*nanoTime*();  
 System.***out***.println(**"\n\nSorting using Merge Sort:"**);  
 **for** (**int** i : numberArray) {  
 System.***out***.print(i + **" "**);  
 }  
 **long** execTimeMS = endTimeMS - startTimeMS;  
 System.***out***.println(**"\nTime Taken for execution using Merge Sort = "** + execTimeMS + **" nanoseconds"**);  
 **return** execTimeMS;  
 }  
  
 *//Heap Sort* **private static long** startHeapSort(**int**[] numberArray) {  
 **long** startTimeHS = System.*nanoTime*();  
 HeapSort heapSort = **new** HeapSort();  
 heapSort.performHeapSort(numberArray);  
 **long** endTimeHS = System.*nanoTime*();  
 System.***out***.println(**"\n\nSorting using Heap Sort:"**);  
 **for** (**int** i : numberArray) {  
 System.***out***.print(i + **" "**);  
 }  
 **long** execTimeHS = endTimeHS - startTimeHS;  
 System.***out***.println(**"\nTime Taken for execution using Heap Sort = "** + execTimeHS + **" nanoseconds"**);  
 **return** execTimeHS;  
 }  
  
 *//In-Place Quick Sort* **private static long** startInPlaceQuickSort(**int**[] numberArray) {  
 **long** startTimeIP = System.*nanoTime*();  
 InPlaceQuickSort inPlaceQuickSort = **new** InPlaceQuickSort();  
 inPlaceQuickSort.performInPlaceQuickSort(numberArray);  
 **long** endTimeIP = System.*nanoTime*();  
 System.***out***.println(**"\n\nSorting using In-Place Quick Sort:"**);  
 **for** (**int** i : numberArray) {  
 System.***out***.print(i + **" "**);  
 }  
 **long** execTimeIP = endTimeIP - startTimeIP;  
 System.***out***.println(**"\nTime Taken for execution using In-Place Quick Sort = "** + execTimeIP + **" nanoseconds"**);  
 **return** execTimeIP;  
 }  
  
 *//Median-of-Three Quick Sort* **private static long** startMedianOfThreeQuickSort(**int**[] numberArray) {  
 **long** startTimeM3 = System.*nanoTime*();  
 MedianOfThreeQuickSort medianOfThreeQuickSort = **new** MedianOfThreeQuickSort();  
 medianOfThreeQuickSort.performSort(numberArray);  
 **long** endTimeM3 = System.*nanoTime*();  
 System.***out***.println(**"\n\nSorting using Median-of-Three Quick Sort:"**);  
 **for** (**int** i : numberArray) {  
 System.***out***.print(i + **" "**);  
 }  
 **long** execTimeM3 = endTimeM3 - startTimeM3;  
 System.***out***.println(**"\nTime Taken for execution using Median-of-Three Quick Sort = "** + execTimeM3 + **" nanoseconds"**);  
 **return** execTimeM3;  
 }  
  
 **private static long** getExecutionTimeIS(**int**[] arrNum1, **int**[] arrNum2, **int**[] arrNum3) {  
 **int**[] arrInsertionSort1 = Arrays.*copyOf*(arrNum1, arrNum1.**length**);  
 **int**[] arrInsertionSort2 = Arrays.*copyOf*(arrNum2, arrNum2.**length**);  
 **int**[] arrInsertionSort3 = Arrays.*copyOf*(arrNum3, arrNum3.**length**);  
 **return** (*startInsertionSort*(arrInsertionSort1) + *startInsertionSort*(arrInsertionSort2) + *startInsertionSort*(arrInsertionSort3)) / 3;  
 }  
  
 **private static long** getExecutionTimeMS(**int**[] arrNum1, **int**[] arrNum2, **int**[] arrNum3) {  
 **int**[] arrMergeSort1 = Arrays.*copyOf*(arrNum1, arrNum1.**length**);  
 **int**[] arrMergeSort2 = Arrays.*copyOf*(arrNum2, arrNum2.**length**);  
 **int**[] arrMergeSort3 = Arrays.*copyOf*(arrNum3, arrNum3.**length**);  
 **return** (*startMergeSort*(arrMergeSort1) + *startMergeSort*(arrMergeSort2) + *startMergeSort*(arrMergeSort3)) / 3;  
 }  
  
 **private static long** getExecutionTimeHS(**int**[] arrNum1, **int**[] arrNum2, **int**[] arrNum3) {  
 **int**[] arrHeapSort1 = Arrays.*copyOf*(arrNum1, arrNum1.**length**);  
 **int**[] arrHeapSort2 = Arrays.*copyOf*(arrNum2, arrNum2.**length**);  
 **int**[] arrHeapSort3 = Arrays.*copyOf*(arrNum3, arrNum3.**length**);  
 **return** (*startHeapSort*(arrHeapSort1) + *startHeapSort*(arrHeapSort2) + *startHeapSort*(arrHeapSort3)) / 3;  
 }  
  
 **private static long** getExecutionTimeIP(**int**[] arrNum1, **int**[] arrNum2, **int**[] arrNum3) {  
 **int**[] arrInPlaceQuickSort1 = Arrays.*copyOf*(arrNum1, arrNum1.**length**);  
 **int**[] arrInPlaceQuickSort2 = Arrays.*copyOf*(arrNum2, arrNum2.**length**);  
 **int**[] arrInPlaceQuickSort3 = Arrays.*copyOf*(arrNum3, arrNum3.**length**);  
 **return** (*startInPlaceQuickSort*(arrInPlaceQuickSort1) + *startInPlaceQuickSort*(arrInPlaceQuickSort2) + *startInPlaceQuickSort*(arrInPlaceQuickSort3)) / 3;  
 }  
  
 **private static long** getExecutionTimeM3(**int**[] arrNum1, **int**[] arrNum2, **int**[] arrNum3) {  
 **int**[] arrMedianOfThreeQuickSort1 = Arrays.*copyOf*(arrNum1, arrNum1.**length**);  
 **int**[] arrMedianOfThreeQuickSort2 = Arrays.*copyOf*(arrNum2, arrNum2.**length**);  
 **int**[] arrMedianOfThreeQuickSort3 = Arrays.*copyOf*(arrNum3, arrNum3.**length**);  
 **return** (*startMedianOfThreeQuickSort*(arrMedianOfThreeQuickSort1) + *startMedianOfThreeQuickSort*(arrMedianOfThreeQuickSort2) + *startMedianOfThreeQuickSort*(arrMedianOfThreeQuickSort3)) / 3;  
 }  
  
 **private static void** sortArrayAscending(**int**[] arrNum1, **int**[] arrNum2, **int**[] arrNum3) {  
 *performAscendingSort*(arrNum1);  
 *performAscendingSort*(arrNum2);  
 *performAscendingSort*(arrNum3);  
 }  
  
 **private static void** sortArrayDescending(**int**[] arrNum1, **int**[] arrNum2, **int**[] arrNum3) {  
 *performDescendingSort*(arrNum1);  
 *performDescendingSort*(arrNum2);  
 *performDescendingSort*(arrNum3);  
 }  
  
 **private static void** performAscendingSort(**int**[] intArray) {  
 InPlaceQuickSort inPlaceQuickSort = **new** InPlaceQuickSort();  
 inPlaceQuickSort.performInPlaceQuickSort(intArray);  
 }  
  
 **private static void** performDescendingSort(**int**[] intArray) {  
 InPlaceQuickSort inPlaceQuickSort = **new** InPlaceQuickSort();  
 inPlaceQuickSort.performInPlaceQuickSort(intArray);  
 **for** (**int** i = 0; i < intArray.**length** / 2; i++) {  
 **int** temp = intArray[i];  
 intArray[i] = intArray[intArray.**length** - i - 1];  
 intArray[intArray.**length** - i - 1] = temp;  
 }  
 }  
}

**Graphs:**

Insertion sort = Red

Merge = Green

Heap = Blue

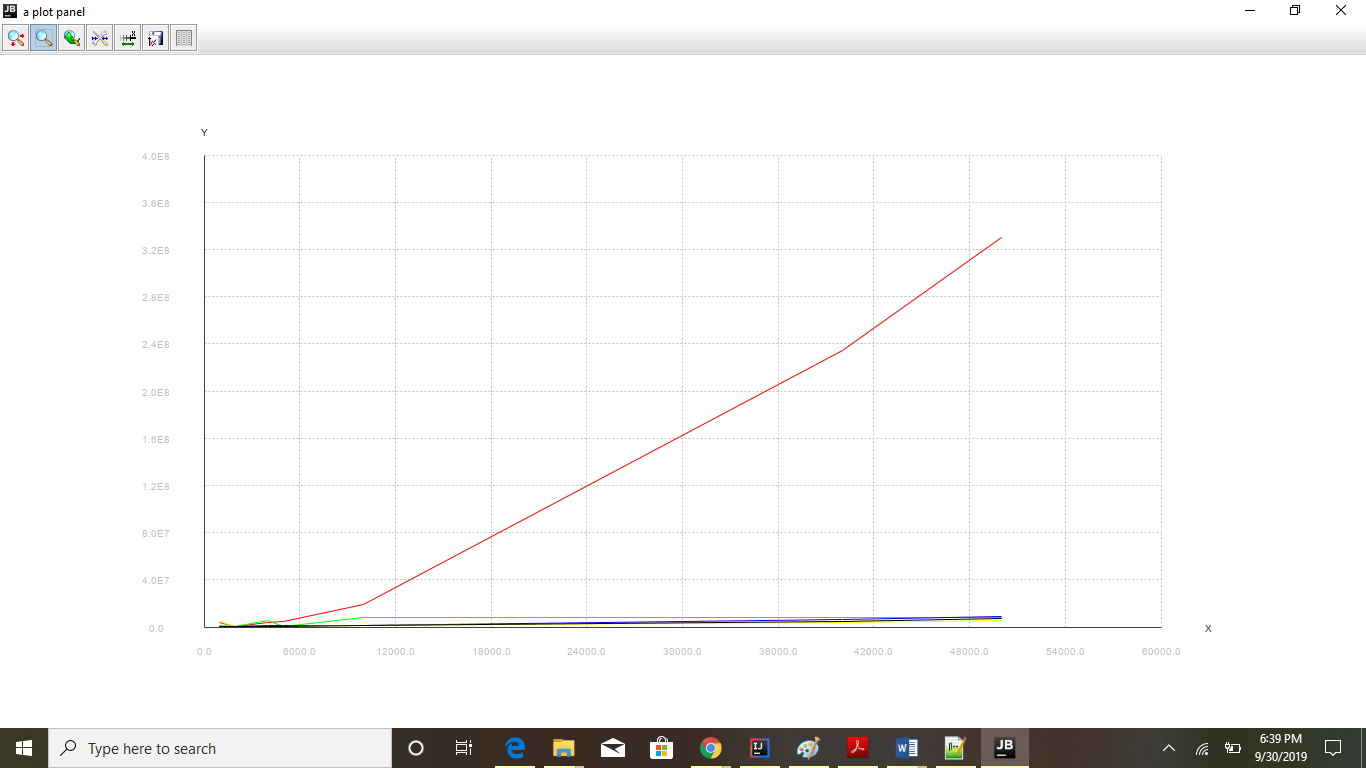
In-place quick = Yellow

Median-of-three quick sort = Black

Also, x-axis represents Array Input Sizes

And y-axis represents Time taken for execution

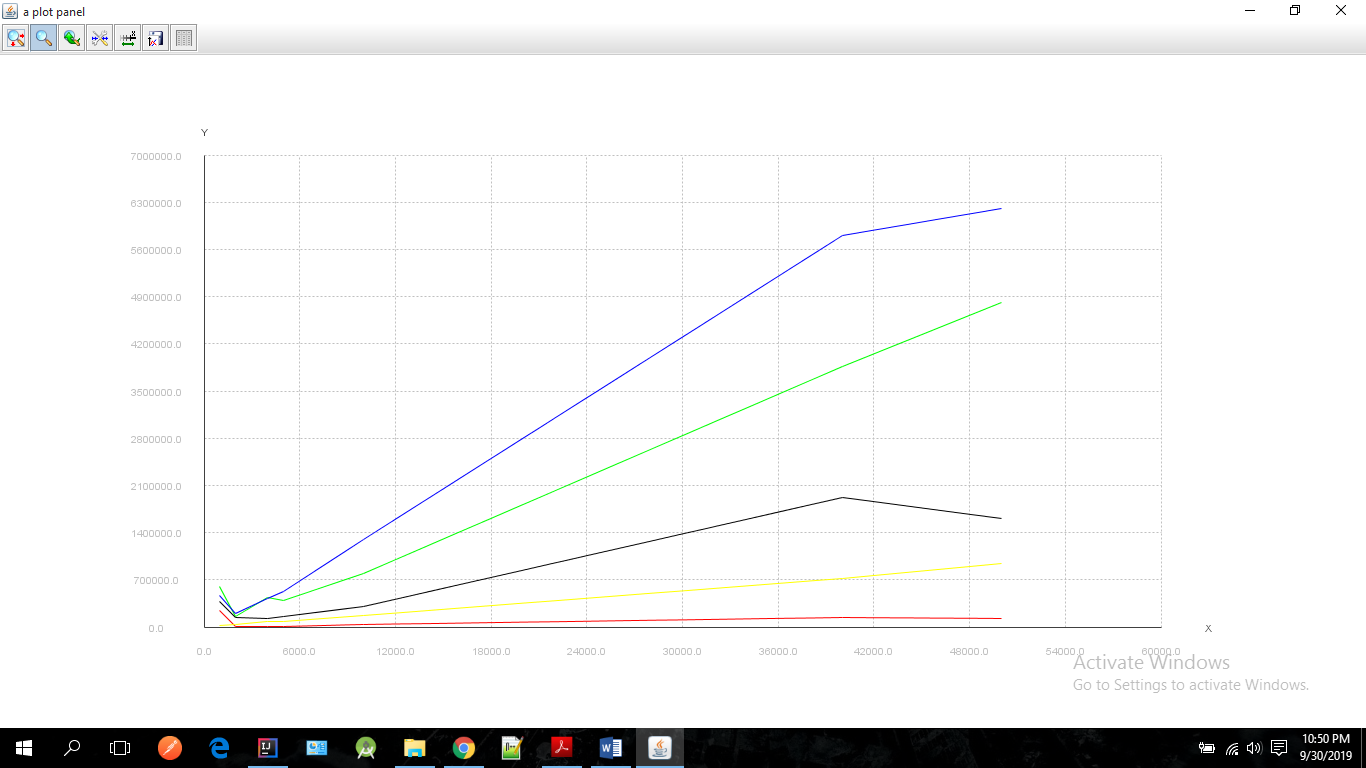
Following graph shows how all the sorting techniques works for different set of data.



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Input Size  S | Insertion Sort | Merge Sort | Heap Sort | Inplace Quick Sort | Modified Quick Sort |
| 1000 | 69670361 | 4050346 | 5908657 | 3709630 | 3249653 |
| 2000 | 288069541 | 9243952 | 17080318 | 6433302 | 12093363 |
| 5000 | 1939442609 | 24926862 | 40146407 | 21301141 | 45137265 |
| 10000 | 8261941845 | 57113218 | 88540210 | 38013277 | 153503571 |
| 20000 | 37761883705 | 131377385 | 248113853 | 84280437 | 600004421 |
| 500000 | 2.73003E+11 | 424418873 | 720590958 | 272483479 | 4152495816 |

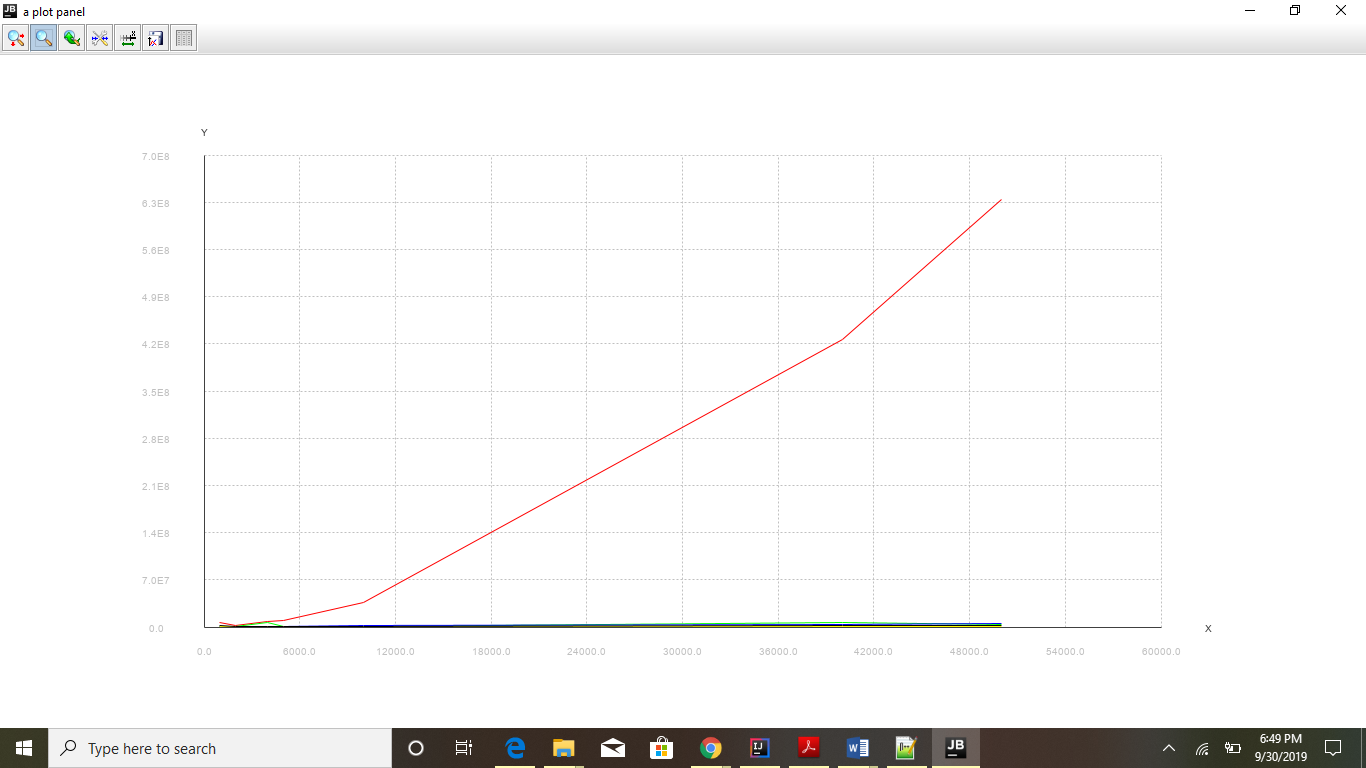
Following two different graphs

Input Array is sorted



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Input Size  S | Insertion Sort | Merge Sort | Heap Sort | Inplace Quick Sort | Modified Quick Sort |
| 1000 | 142804 | 3425152 | 7581599 | 2302866 | 223511 |
| 2000 | 374540 | 6636046 | 12192781 | 244381 | 3274945 |
| 5000 | 668992 | 17624086 | 33128619 | 8985280 | 33128619 |
| 10000 | 1290381 | 39314249 | 73812827 | 19361697 | 2374936 |
| 20000 | 3200715 | 80915943 | 1.59E+08 | 40830405 | 6039330 |
| 500000 | 6734333 | 2.13E+08 | 4.37E+08 | 1.1E+08 | 23804062 |

Input Array is reversely Sorted



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Input Size  S | Insertion Sort | Merge Sort | Heap Sort | Inplace Quick Sort | Modified Quick Sort |
| 1000 | 1.26E+08 | 2845297 | 1220881 | 1.22E+08 | 7311823 |
| 2000 | 5.1E+08 | 6251635 | 15220088 | 2789470 | 5.27E+08 |
| 5000 | 3.46E+09 | 18521525 | 50118669 | 8935005 | 3.56E+09 |
| 10000 | 1.68E+10 | 50879677 | 22515531 | 8.44E+09 | 1.31E+08 |
| 20000 | 6.96E+10 | 97612983 | 46518738 | 2.72E+10 | 3.13E+08 |