Project 1: Comparison Based Sorting Algorithms

Project Structure:

- Our code contains 8 classes.
- **Insertion_Sort.java** and **Merge_Sort.java** contains code to perform insertion and merge sort.
- **Quick_Sort_Inplace.java** contains code for in place quick sort.
- **Quick_Median_of_3.java** contains code to implement quicksort using median of 3 approach.
- **Quick_Small_Subproblem.java** contains code to implement quick sort if the input array length is greater than 10. This class calls insertion sort, if the array length is less than or equal to 10.
- **Performance.java** takes the size of the array as input and calculates the time take to execute each sorting algorithm.
- **Sorted_Reversesorted.java** contains code to measure the time taken to sort a sorted array and an inverted sorted array.
- **Sorting_Algorithm.java** contains the main class which takes the size and range of the input array and generates an array of random members. It prompts the user to select the sorting algorithm to sort this random number array. It also computes the time taken to sort the random array selected by the user. It also displays the sorted array.

Observations:

We have recorded the execution time for all the sorting algorithms in a table and computed the average values. We also plotted graphs by taking **input size on X axis** and time **consumed on the Y axis**.

We have recorded time taken for execution of different random inputs for 5 times and took the average.

Input Size = 500

	1	2	3	4	5	Average
Insertion	2.32023	3.09502	2.16240	2.36931	2.18505	2.42640
Merge	1.34796	1.20523	1.86524	1.79501	1.34909	1.51251
Quick Inplace	1.96341	1.99815	1.86487	2.27907	2.10048	2.0412
Quick Median of 3	4.29535	1.54657	4.06994	3.03310	2.89151	3.16729
Quick with Insertion	4.10014	3.67272	2.48108	4.2727	3.90456	3.68624

Input Size = 1000

	1	2	3	4	5	Average
Insertion	5.50248	6.21988	4.97537	5.91857	6.26783	5.77683
Merge	2.65627	2.60341	2.52752	2.57585	3.19319	2.71125
Quick Inplace	5.09356	5.91479	3.44731	4.19038	5.46434	4.82208
Quick Median of 3	4.90174	4.87041	4.68275	4.78923	5.36466	4.92176
Quick with Insertion	7.26540	8.19274	5.50965	6.58500	6.36336	6.78323

Input Size = 2000

	1	2	3	4	5	Average
Insertion	6.74283	7.89067	10.9505	6.38715	6.67751	7.72975
Merge	7.66526	8.50273	8.21048	10.1293	8.95658	8.69288
Quick Inplace	12.4503	15.3456	9.34209	13.5491	9.18049	11.9735
Quick Median of 3	9.74082	16.8363	9.94320	16.9699	10.5454	12.8071
Quick with Insertion	8.55106	22.5725	8.94979	19.9604	8.64055	13.7348

Input Size = 4000

	1	2	3	4	5	Average
Insertion	9.20201	17.6556	11.6094	10.34	13.73	12.5074
Merge	36.2167	33.4872	34.89	25.88	24.56	31.0068
Quick Inplace	35.6024	31.2795	50.23	29.63	29.21	35.1903
Quick Median of 3	21.7074	20.8595	35.65	28.46	23.24	25.9834
Quick with Insertion	20.0495	19.5209	34.38	22.68	15.01	22.3280

Input Size = 5000

	1	2	3	4	5	Average
Insertion	10.34	14.17	14.97	22.48	18.37	16.066
Merge	14.54	16.69	18.27	19.01	19.16	17.534
Quick Inplace	19.72	18.9	27.16	21.84	24.23	22.37
Quick Median of 3	19.95	18.07	25.9	21.81	29.28	23.002
Quick with Insertion	20.41	16.75	34.53	29.64	31.78	26.622

Input Size = 10000

	1	2	3	4	5	Average
Insertion	36.65	27.15	23.5	24.83	25.24	27.474
Merge	12.89	15.87	19.51	15.4	22.29	17.192
Quick Inplace	24.43	22.04	22.96	18.84	26.99	23.052
Quick Median of 3	15.93	17.58	19.26	22.28	19.69	18.948
Quick with Insertion	14.31	15.38	33.27	30.53	28.9	24.478

Input Size = 20000

	1	2	3	4	5	Average
Insertion	27.92	29.21	24.7	32.75	40.22	30.96
Merge	24.52	23.1	37.86	46.98	38.97	34.286
Quick Inplace	13.73	13.33	18.34	23.46	14.96	16.764
Quick Median of 3	39.75	21.63	37.4	39.91	39.32	35.602
Quick with Insertion	20.77	23.68	30.86	35.85	31.43	28.518

Input Size = 30000

	1	2	3	4	5	Average
Insertion	14.72	12.22	12.76	12.7	12.02	12.884
Merge	47	51.4	72.67	72.34	85.31	65.744
Quick Inplace	34.18	33.11	33.69	48.64	39.47	37.818
Quick Median of 3	53.64	68.4	104.18	75.24	103.85	81.062

Quick with Insertion	48.17	61.59	82.1	75.24	68.97	67.214

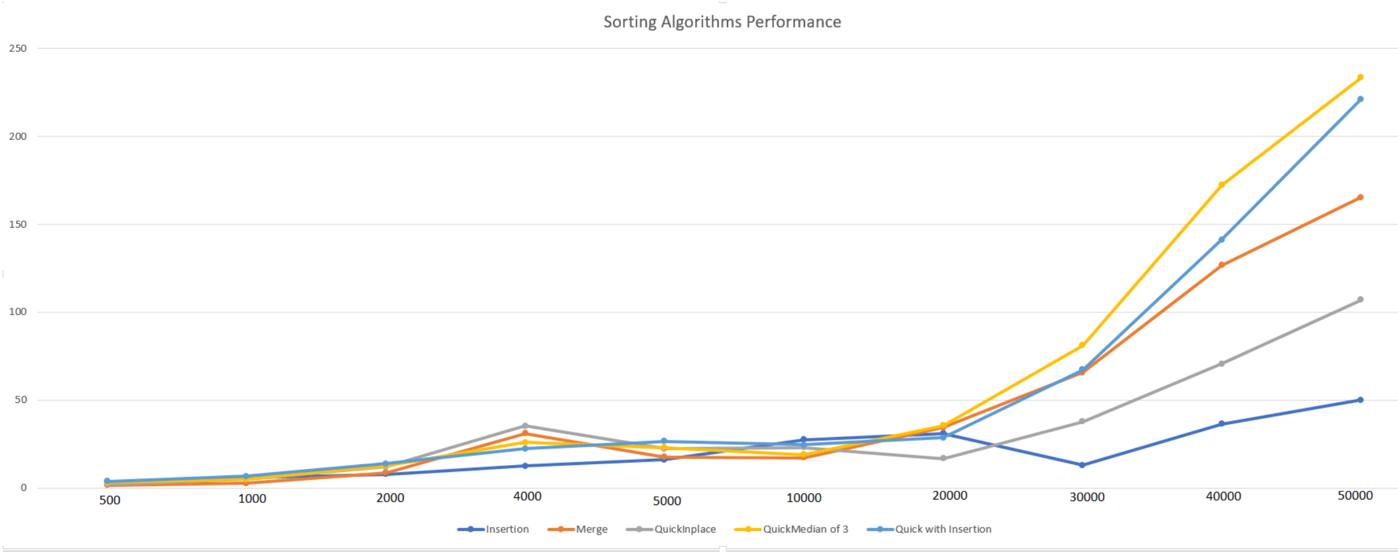
Input Size = 40000

	1	2	3	4	5	Average
Insertion	37.58	31.08	57.28	24.82	31.68	36.488
Merge	121.41	109.34	129.72	135.89	136.89	126.65
Quick Inplace	60.78	71.41	71.69	91.39	57.6	70.574
Quick Median of 3	213.44	154.87	238.2	142.27	112.73	172.302
Quick with Insertion	141.08	141.01	158.32	128.6	136.69	141.14

Input Size = 50000

	1	2	3	4	5	Average
Insertion	41.58	48.5	64.18	60.63	35.78	50.134
Merge	184.56	177.31	141.3	155.63	168.09	165.378
Quick Inplace	132.12	86.37	91.13	97.04	128.9	107.112
Quick Median of 3	304.35	185.04	253.48	198.06	227.06	233.598
Quick with Insertion	207.65	242.58	214.28	237.22	204.16	221.178

Graph for the averagetime taken for the algorithms.(Performance_Graph.png)



Performance for Sorted and Inversely Sorted Inputs

We have tested performance for sorted and inverted sorted input.

Results: (in micro seconds)

We have observed that Merge sort performed well in these cases.

Sorted Input:

Time consumption for Insertion sort (sorted) is: 860883

Time consumption for Merge sort (sorted) is: 1705154

Time consumption for Inplace Quick sort (sorted) is: 3585882

Time consumption for Quick sort with median of three (sorted) is: 5545903

Time consumption for Quick sort with insertion sort (sorted) is: 3476006

Inversely Sorted Input:

Time consumption for Insertion sort (reversely sorted) is: 591291

Time consumption for Merge sort (reversely sorted) is: 539185

Time consumption for Inplace Quick sort (reversely sorted) is: 570901

Time consumption for Quick sort with median of three (reversely sorted) is: 1222228

Time consumption for Quick sort with insertion sort (reversely sorted) is: 77630

Code:

Sorting_Algorithm.java

```java
import java.util.Random;
import java.util.Scanner;

public class Sorting_Algorithm {
    public static void main(String args[]){
        int range, n;
        Random rand = new Random();
        Scanner scan = new Scanner(System.in);
        System.out.println("enter the size of the random array you want to sort");
        n = scan.nextInt();
        System.out.println("enter the range of the elements");
        range = scan.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = rand.nextInt(range);
        }
        long begin = System.nanoTime();
        System.out.println("Elements before sorting");
        for(int i:arr)
            System.out.print(i+" "); //printing array before sorting

        System.out.println("\n\nEnter the sorting algorithm you want to implement (1-
5)\n1.Insertion Sort\n2.Merge Sort\n3.Inplace Quick Sort\n4.Median of three Quick
Sort\n5.Quick Sort with Insertion Sort");
        Scanner inp = new Scanner(System.in);
        int sort_kind = inp.nextInt();
            if (sort_kind == 1) {
                Insertion_Sort IS = new Insertion_Sort();
                IS.sort(arr);
            } else if (sort_kind == 2) {
                Merge_Sort MS = new Merge_Sort();
                MS.sort(arr, 0, arr.length - 1);
            } else if (sort_kind == 3) {
                Quick_Sort_Inplace QSIP = new Quick_Sort_Inplace();
                QSIP.sort(arr, 0, arr.length - 1);
            } else if (sort_kind == 4) {
                Quick_Median_Of_Three QMOT = new Quick_Median_Of_Three();
                QMOT.sort(arr,0,arr.length-1);
            }
            else if (sort_kind == 5) {
                Quick_Small_Subproblem QSS = new Quick_Small_Subproblem();
                QSS.sort(arr,0,arr.length-1);
            }
            else return;

        System.out.println("\nElements after sorting");
        for(int i:arr)
            System.out.print(i+" ");//printing the array after sorting
        System.out.println();

        long end = System.nanoTime();
        long totalTime = end - begin; //computes the total time consumed to run the code
        System.out.println("Total time in micro seconds:"+totalTime/1000000);
    }
}
```

Insertion_Sort.java

```
import java.util.Random;
import java.util.Scanner;

public class Insertion_Sort {

    public void sort(int inputArray[]) {
        for (int j = 1; j < inputArray.length; j++) {
            int key = inputArray[j];
            int i = j - 1;
            while (i >= 0 && inputArray[i] > key) //If the elements in the array are greater than key, we will move them to the right by 1 position
            {
                inputArray[i + 1] = inputArray[i]; //moving to right by one position
                i = i - 1;
            }
            inputArray[i + 1] = key; // Inserting the key at correct position
        }
    }
}
```

Merge_Sort.java

```
import java.util.Random;
import java.util.Scanner;

public class Merge_Sort {
    public void sort(int inputArray[],int left, int right) {
        if (left < right) {
            int center = left + (right - left) / 2;
            sort(inputArray,left, center); //left subarray
            sort(inputArray,center + 1, right); //right subarray
            merge(inputArray,left, center, right); //merging the two halfs
        }
    }
    private void merge(int inputArray[],int left, int center, int right) {
        int[] temp = new int[inputArray.length];
        for (int i = left; i <= right; i++) {
            temp[i] = inputArray[i];
        }
        int i = left;
        int j = center + 1;
        int k = left;
        while (i <= center && j <= right) {
            if (temp[i] <= temp[j]) {
                inputArray[k] = temp[i];
                i++;
                k++;
            } else {
                inputArray[k] = temp[j];
                j++;
                k++;
            }
        }
        while (i <= center) {  //rest of the elements of left subarray
            inputArray[k] = temp[i];
            k++;
            i++;
        }
        while (j <= right) {
            inputArray[k] = temp[j]; //rest of the elements of the right subarray
            k++;
            j++;
        }

    }
}
```

Quick_Sort_Inplace.java

```
import java.util.Arrays;
import java.util.Random;
import java.util.Scanner;

public class Quick_Sort_Inplace
{
    public void sort(int inputArray[], int leftIndex, int rightIndex) {
        if (rightIndex > leftIndex) {
            int i = leftIndex, j = rightIndex;
            int tempSwap=0;
            // taking the last element as Pivot
            int pivot = inputArray[rightIndex];
            do {
                //Increment i value until inputArray[i] value greater than pivot
                while (inputArray[i] < pivot)
                    i++;
                //Decrement j value until inputArray[j] value is less than pivot
                while (inputArray[j] > pivot)
```

```
                    j--;
                // if i index value is less than or equal to j index value then SWAP the elements
                if (i <= j) {
                    tempSwap = inputArray[j];
                    inputArray[j] = inputArray[i];
                    inputArray[i] = tempSwap;
                    i++;
                    j--;
                }
            } while (i <= j); // continue until index i value is less than j value
            if (leftIndex < j) {
                sort(inputArray, leftIndex, j); // sorts the left subarray before the pivot
            }
            if (i < rightIndex) {
                sort(inputArray, i, rightIndex); //sorts the right subarray after the pivot
            }
        }
    }
}
```

Quick_Median_Of_Three.java

```
import java.util.Random;
import java.util.Scanner;

public class Quick_Median_Of_Three {
    public static void sort(int[] inputArray, int left, int right) {
        if (right-left < 3) {
            Insertion_Sort IS = new Insertion_Sort(); //insertion sort if array length is <3
            IS.sort(inputArray);
        } else {
            double median = medianOfThree(inputArray, left, right);
            int partition = partition(inputArray, left, right, median);
            sort(inputArray, left, partition - 1);
            sort(inputArray, partition + 1, right);
        }
    }

    public static int medianOfThree(int[] inputArray, int left, int right) { //median of three logic
        int center = (left + right) / 2;

        if (inputArray[left] > inputArray[center])
            swap(inputArray, left, center);

        if (inputArray[left] > inputArray[right])
            swap(inputArray, left, right);

        if (inputArray[center] > inputArray[right])
            swap(inputArray, center, right);

        swap(inputArray, center, right - 1); //swapping center with right-1 position
        return inputArray[right - 1];
    }
    public static void swap(int[] inputArray, int a, int b) {
        int temp = inputArray[a];
        inputArray[a] = inputArray[b];
        inputArray[b] = temp;
    }

    public static int partition(int[] inputArray, int left, int right, double pivot) {
        int leftTemp = left;
        int rightTemp = right - 1;

        while (true) {
            while (inputArray[++leftTemp] < pivot)
                ;
            while (inputArray[--rightTemp] > pivot)
                ;
            if (leftTemp >= rightTemp)
                break;
            else
                swap(inputArray, leftTemp, rightTemp);
        }
        swap(inputArray, leftTemp, right - 1);
        return leftTemp;
    }
}
```

Quick_Small_Subproblem.java

```
class Quick_Small_Subproblem
{
    public void sort(int[] inputArray, int left, int right)
    {
        if (left < right)
        {
            if (left-right <= 10) //If size is less than or equal to 10, call insertion sort
            {
                Insertion_Sort IS = new Insertion_Sort();
                IS.sort(inputArray);
            }
            else
            {
                int partition = this.partition(inputArray, left, right);
                this.sort(inputArray, left, partition - 1);
                this.sort(inputArray, partition + 1, right);
            }
        }
    }
    private int partition(int[] inputArray, int left, int right)
    {
        int leftTemp = left;
        int rightTemp = right;
        int pivot = inputArray[left];
        while (leftTemp < rightTemp)
        {
            if (inputArray[leftTemp] < pivot)
            {
                leftTemp++;
                continue;
            }
            if (inputArray[rightTemp] > pivot)
            {
                rightTemp--;
                continue;
            }
            int tmp = inputArray[leftTemp];
            inputArray[leftTemp] = inputArray[rightTemp];
            inputArray[rightTemp] = tmp;
            leftTemp++;
        }
        return leftTemp;
    }
}
```

Performance.java

```
import java.math.BigInteger;
import java.util.Random;
import java.util.Scanner;
public class Performance {
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the size of the array");
        int size = sc.nextInt();
        Random rand = new Random();
        int[] arr = new int[size];
        for (int i = 0; i < size; i++) {
            arr[i] = rand.nextInt(size);
        }
        long begin = System.nanoTime();
        System.out.println("Elements before sorting");
        for(int i:arr){
            System.out.print(i+" ");
        }; //printing array before sorting

        long Ins_begin_time = System.nanoTime();
        Insertion_Sort obji = new Insertion_Sort();
        obji.sort(arr);
        long Ins_end_time = System.nanoTime();
        long Ins_total_time = Ins_end_time - Ins_begin_time;
        System.out.println("\nTime consumption for Insertion sort is:"+Ins_total_time);

        long Mer_begin_time = System.nanoTime();
        Merge_Sort objm = new Merge_Sort();
        objm.sort(arr,0,arr.length-1);
        long Mer_end_time = System.nanoTime();
        long Mer_total_time = Mer_end_time - Mer_begin_time;
        System.out.println("Time consumption for Merge sort is:"+Mer_total_time);

        long Quick_begin_time = System.nanoTime();
        Quick_Sort_Inplace objq = new Quick_Sort_Inplace();
        objq.sort(arr,0,arr.length-1);
        long Quick_end_time = System.nanoTime();
        long Quick_total_time = Quick_end_time - Quick_begin_time;
        System.out.println("Time consumption for Inplace Quick sort is:"+Quick_total_time);

        long QMed_begin_time = System.nanoTime();
        Quick_Median_Of_Three objm3 = new Quick_Median_Of_Three();
        objm3.sort(arr,0,arr.length-1);
        long QMed_end_time = System.nanoTime();
        long QMed_total_time = QMed_end_time - QMed_begin_time;
        System.out.println("Time consumption for Quick sort with median of three
is:"+QMed_total_time);

        long QIns_begin_time = System.nanoTime();
        Quick_Small_Subproblem objqss = new Quick_Small_Subproblem();
        objqss.sort(arr,0,arr.length-1);
        long QIns_end_time = System.nanoTime();
        long QIns_total_time = QIns_end_time - QIns_begin_time;
        System.out.println("Time consumption for Quick sort with insertion sort is:"+
QIns_total_time);
    }
}
```

Sorted_ReverseSorted.java

```
import java.util.Arrays;
import java.util.Collections;

public class Sorted_ReverseSorted {
  public static void main(String args[]) {
    int arr[] = {2, 3, 6, 7, 8, 9, 10, 12, 12, 14, 16, 17, 18, 19, 21, 21, 23, 24, 24, 25, 27, 28, 28, 28, 28,
28, 29, 31, 32, 32, 34, 34, 39, 39, 39, 40, 40, 41, 44, 46, 49, 49, 50, 51, 51, 54, 56, 58, 59, 59, 60, 60,
63, 64, 64, 65, 65, 65, 66, 67, 70, 70, 71, 72, 74, 74, 76, 76, 76, 76, 78, 79, 81, 82, 83, 84, 85, 86, 87,
87, 87, 88, 88, 89, 90, 90, 90, 95, 98, 99, 100, 100, 100, 101, 101, 103, 103, 106, 106, 108, 108, 110,
111, 112, 113, 114, 115, 116, 116, 117, 118, 119, 119, 120, 121, 122, 122, 123, 123, 124, 124, 124,
128, 128, 131, 132, 132, 135, 135, 136, 144, 144, 145, 146, 147, 147, 149, 150, 151, 151, 151, 151,
152, 154, 154, 154, 154, 155, 156, 156, 159, 160, 161, 162, 164, 164, 164, 164, 165, 166, 167, 168,
168, 169, 171, 172, 173, 173, 174, 174, 174, 175, 175, 177, 179, 180, 181, 183, 184, 185, 187, 189,
189, 189, 190, 190, 190, 194, 195, 195, 195, 195, 195, 197, 197, 198, 199, 200, 200, 202, 202, 203,
203, 204, 204, 206, 206, 207, 208, 208, 211, 212, 212, 212, 213, 213, 214, 216, 217, 218, 218, 218,
220, 223, 224, 225, 225, 225, 227, 227, 227, 228, 229, 229, 230, 231, 232, 234, 234, 234, 234, 235,
235, 238, 240, 242, 242, 244, 244, 245, 248, 248, 250, 251, 251, 251, 252, 253, 255, 255, 255, 256,
257, 257, 259, 260, 262, 262, 263, 263, 264, 266, 268, 268, 269, 269, 271, 271, 272, 272, 272, 273,
274, 274, 275, 276, 280, 281, 283, 286, 288, 288, 289, 289, 290, 291, 294, 295, 295, 298, 301, 301,
303, 304, 304, 305, 307, 308, 308, 309, 309, 309, 309, 310, 311, 313, 314, 315, 316, 316, 317, 317,
318, 318, 320, 320, 320, 321, 321, 322, 323, 324, 325, 326, 327, 327, 329, 331, 333, 334, 335, 337,
338, 339, 340, 340, 340, 342, 342, 343, 344, 344, 345, 345, 345, 346, 346, 347, 348, 348, 349, 351,
351, 352, 352, 352, 355, 356, 358, 358, 361, 361, 361, 362, 363, 364, 365, 365, 365, 366, 366, 367,
368, 368, 368, 369, 370, 371, 371, 372, 373, 375, 378, 378, 379, 380, 380, 380, 384, 386, 389, 390,
391, 391, 392, 392, 394, 394, 396, 397, 400, 400, 400, 401, 402, 406, 408, 415, 417, 419, 419, 421,
422, 422, 424, 424, 426, 428, 429, 430, 430, 431, 431, 434, 435, 437, 439, 439, 443, 444, 446, 447,
449, 451, 451, 452, 453, 453, 454, 456, 456, 458, 458, 459, 460, 460, 461, 463, 463, 464, 464, 465,
465, 467, 467, 468, 470, 471, 472, 472, 473, 473, 474, 474, 475, 475, 476, 478, 479, 479, 483, 484,
485, 485, 486, 488, 489, 490, 491, 492, 492, 492, 492, 493, 494, 494, 494, 497, 498, 499};
        int brr[] = {499, 498, 497, 494, 494, 494, 493, 492, 492, 492, 492, 491, 490, 489, 488, 486, 485,
485, 484, 483, 479, 479, 478, 476, 475, 475, 474, 474, 473, 473, 472, 472, 471, 470, 468, 467, 467,
465, 465, 464, 464, 463, 463, 461, 460, 460, 459, 458, 458, 456, 456, 454, 453, 453, 452, 451, 451,
449, 447, 446, 444, 443, 439, 439, 437, 435, 434, 431, 431, 430, 430, 429, 428, 426, 424, 424, 422,
422, 421, 419, 419, 417, 415, 408, 406, 402, 401, 400, 400, 400, 397, 396, 394, 394, 392, 392, 391,
391, 390, 389, 386, 384, 380, 380, 380, 379, 378, 378, 375, 373, 372, 371, 371, 370, 369, 368, 368,
368, 367, 366, 366, 365, 365, 365, 364, 363, 362, 361, 361, 361, 358, 358, 356, 355, 352, 352, 352,
351, 351, 349, 348, 348, 347, 346, 346, 345, 345, 345, 344, 344, 343, 342, 342, 340, 340, 340, 339,
338, 337, 335, 334, 333, 331, 329, 327, 327, 326, 325, 324, 323, 322, 321, 321, 320, 320, 320, 318,
318, 317, 317, 316, 316, 315, 314, 313, 311, 310, 309, 309, 309, 309, 308, 308, 307, 305, 304, 304,
303, 301, 301, 298, 295, 295, 294, 291, 290, 289, 289, 288, 288, 286, 283, 281, 280, 276, 275, 274,
274, 273, 272, 272, 272, 271, 271, 269, 269, 268, 268, 266, 264, 263, 263, 262, 262, 260, 259, 257,
257, 256, 255, 255, 255, 253, 252, 251, 251, 251, 250, 248, 248, 245, 244, 244, 242, 242, 240, 238,
235, 235, 234, 234, 234, 234, 232, 231, 230, 229, 229, 228, 227, 227, 227, 225, 225, 225, 224, 223,
220, 218, 218, 218, 217, 216, 214, 213, 213, 212, 212, 212, 211, 208, 208, 207, 206, 206, 204, 204,
203, 203, 202, 202, 200, 200, 199, 198, 197, 197, 195, 195, 195, 195, 195, 194, 190, 190, 190, 189,
189, 189, 187, 185, 184, 183, 181, 180, 179, 177, 175, 175, 174, 174, 174, 173, 173, 172, 171, 169,
168, 168, 167, 166, 165, 164, 164, 164, 164, 162, 161, 160, 159, 156, 156, 155, 154, 154, 154, 154,
152, 151, 151, 151, 151, 150, 149, 147, 147, 146, 145, 144, 144, 136, 135, 135, 132, 132, 131, 128,
128, 124, 124, 124, 123, 123, 122, 122, 121, 120, 119, 119, 118, 117, 116, 116, 115, 114, 113, 112,
111, 110, 108, 108, 106, 106, 103, 103, 101, 101, 100, 100, 100, 99, 98, 95, 90, 90, 90, 89, 88, 88, 87,
87, 87, 86, 85, 84, 83, 82, 81, 79, 78, 76, 76, 76, 76, 74, 74, 72, 71, 70, 70, 67, 66, 65, 65, 65, 64, 64,
63, 60, 60, 59, 59, 58, 56, 54, 51, 51, 50, 49, 49, 46, 44, 41, 40, 40, 39, 39, 39, 34, 34, 32, 32, 31, 29,
28, 28, 28, 28, 28, 27, 25, 24, 24, 23, 21, 21, 19, 18, 17, 16, 14, 12, 12, 10, 9, 8, 7, 6, 3, 2};

        long Ins_begin_time = System.nanoTime();
        Insertion_Sort obji = new Insertion_Sort();
        obji.sort(arr);
        long Ins_end_time = System.nanoTime();
        long Ins_total_time = Ins_end_time - Ins_begin_time;
        System.out.println("\nTime consumption for Insertion sort (sorted) is:" + Ins_total_time);

        long Mer_begin_time = System.nanoTime();
        Merge_Sort objm = new Merge_Sort();
        objm.sort(arr, 0, arr.length - 1);
        long Mer_end_time = System.nanoTime();
        long Mer_total_time = Mer_end_time - Mer_begin_time;
        System.out.println("Time consumption for Merge sort (sorted) is:" + Mer_total_time);

        long Quick_begin_time = System.nanoTime();
        Quick_Sort_Inplace objq = new Quick_Sort_Inplace();
        objq.sort(arr, 0, arr.length - 1);
        long Quick_end_time = System.nanoTime();
        long Quick_total_time = Quick_end_time - Quick_begin_time;
        System.out.println("Time consumption for Inplace Quick sort (sorted) is:" +
Quick_total_time);

        long QMed_begin_time = System.nanoTime();
        Quick_Median_Of_Three objm3 = new Quick_Median_Of_Three();
        objm3.sort(arr, 0, arr.length - 1);
        long QMed_end_time = System.nanoTime();
        long QMed_total_time = QMed_end_time - QMed_begin_time;
        System.out.println("Time consumption for Quick sort with median of three (sorted) is:" +
QMed_total_time);
```

```
        long QIns_begin_time = System.nanoTime();
        Quick_Small_Subproblem objqss = new Quick_Small_Subproblem();
        objqss.sort(arr, 0, arr.length - 1);
        long QIns_end_time = System.nanoTime();
        long QIns_total_time = QIns_end_time - QIns_begin_time;
        System.out.println("Time consumption for Quick sort with insertion sort (sorted) is:" +
QIns_total_time);



        long Ins_begin_time2 = System.nanoTime();
        Insertion_Sort obji2 = new Insertion_Sort();
        obji2.sort(brr);
        long Ins_end_time2 = System.nanoTime();
        long Ins_total_time2 = Ins_end_time2 - Ins_begin_time2;
        System.out.println("\nTime consumption for Insertion sort (reversly sorted) is:" +
Ins_total_time2);

        long Mer_begin_time2 = System.nanoTime();
        Merge_Sort objm2 = new Merge_Sort();
        objm2.sort(brr, 0, brr.length - 1);
        long Mer_end_time2 = System.nanoTime();
        long Mer_total_time2 = Mer_end_time2 - Mer_begin_time2;
        System.out.println("Time consumption for Merge sort (reversly sorted) is:" +
Mer_total_time2);

        long Quick_begin_time2 = System.nanoTime();
        Quick_Sort_Inplace objq2 = new Quick_Sort_Inplace();
        objq2.sort(brr, 0, brr.length - 1);
        long Quick_end_time2 = System.nanoTime();
        long Quick_total_time2 = Quick_end_time2 - Quick_begin_time2;
        System.out.println("Time consumption for Inplace Quick sort (reversly sorted) is:" +
Quick_total_time2);

        long QMed_begin_time2 = System.nanoTime();
        Quick_Median_Of_Three objm32 = new Quick_Median_Of_Three();
        objm32.sort(brr, 0, brr.length - 1);
        long QMed_end_time2 = System.nanoTime();
        long QMed_total_time2 = QMed_end_time2 - QMed_begin_time2;
        System.out.println("Time consumption for Quick sort with median of three (reversly sorted)
is:" + QMed_total_time2);

        long QIns_begin_time2 = System.nanoTime();
        Quick_Small_Subproblem objqss2 = new Quick_Small_Subproblem();
        objqss2.sort(brr, 0, brr.length - 1);
        long QIns_end_time2 = System.nanoTime();
        long QIns_total_time2 = QIns_end_time2 - QIns_begin_time2;
        System.out.println("Time consumption for Quick sort with insertion sort (reversly sorted)
is:" + QIns_total_time2);
    }
}
```

Conclusion/ Understanding:

We have observed that Merge Sort performed well overall when we compare all the algorithms' execution time.

Also, the Inplace quicksort did a better job when compared to the median of the three approach.

Insertion sort performed well when the input size is less.

Median of three Quick sort performed worst compared to all other algorithms.

Submitted by:

Manideep Reddy Nukala (801060367)

Sai Charan Reddy Vallapureddy (801083895)