Analyzation of Sorting Algorithms, Lab 5, Java II

Terry Speicher *Student, EPCC,* Jason Ivey, *Member, CSIT,*

Abstract—Sorting has become a sub-field of computer science. The ability to take a set of data and arrange it into a specific sequence of ascending or descending values has created a multitude of algorithms and analyses. Our challenge was to create a class from which we could run at least 6 different sorting algorithms coded in Java and compare the performance of each on a given set of data.

Keywords—Computer Society, IEEEtran, journal, LaTeX, Force Push, nanosecond unreliability.

1 Introduction

OUR team created a class that contained 6 different sorting routines. We also created a separate class to handle the acquisition of the data to be sorted, the invoking of the Sort class to access the individual sort methods, and the recording of the time that it would take to perform each task. We also wanted to be able to provide different sizes of data sets to be able to compare sort times for varying values of n.

tjs
November 5, 2016

1.1 Tasking the Sorts

Our task was not to code each sorting routine from scratch, although we did so with the Bubble sort and our own Force Push sort. Our task was, instead, to assemble the sorts from other documented sources such as the textbook (in the case of the Quick Sort) or the Internet as noted in each method. Once the sorting routines were gathered and assembled, a methodology had to be developed to provide fair and accurate testing and recording. Each sorting algorithm was structured to receive a reference to an array of type Point. The class Point was defined in a separate point.java class file and consisted of two double values representing the (x,y) Cartesian graphing pair. The array would be sorted in ascending order based on the x coordinate. No return type was required since the reference passed as an actual parameter was a reference to the array from the invoking class. Thus, the original array was the one being sorted. System time was recorded in nanoseconds before and after each sorting routine was called. The difference of the two times was logged as the time that it took that sort algorithm to perform a sort on the array with n elements.

1.2 Methodology

Our methodology for testing the different sorting algorithms was to create a sort

- *T. Speicher is a student in the Computer Science department of El Paso Community College and the University of Texas, El Paso. He is currently Owner of Timely Enterprises, Inc., a professional outsourced IT support provider.*
  E-mail: Terry@TimelyEP.com
- *Jason Ivey is a undergraduate in the field of Computer Science and member of the CSIT, El Paso division.*

Manuscript received November 5, 2016; revision is scheduled for summer 2021.

testing class that would have an array of integers representing the different sizes of arrays (different size n arrays) to be sorted. This gave us great flexibility in deciding the different sizes of the data sets. Given a file with 100,000 test elements, this array of integers was used to create sub arrays from the full list.

Each sort was invoked given an array of sizes varying from 10 elements to 100,000 elements. Each sort was timed and that time was recorded. Because run time of a program can be influenced by other tasks the computer may be performing at the same time, we ran each test 10 times and took the average time for each test case.

We noticed that there were anomalies with the results when the test cases used

$$n < 10$$

, regardless of the sort routine called. In fact, the first test of each routine resulted in times that were higher than the second run of the same routine even when the same data was being used. To counter this anomaly, we disregarded the results of the first few tests i.e.

$$n = 2, n = 3, n = 4, ...n = 9$$

.

1.3 Hypothesis

We were working on a hypothesis that an

$$O(n^2)$$

sort might have better real time performance on lower n elements that an

$$O(nlogn)$$

sort. Thus, we structured our test cases to cover more tests of

$$n < 100$$

However, our tests did not show a significant benefit for using a non-recursive or

$$O(n^2)$$

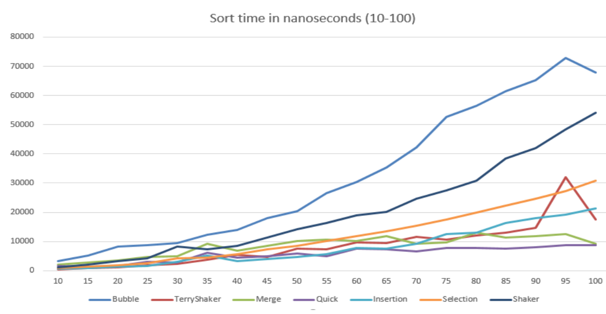
sort for lower values of n.

1.4 Graphs

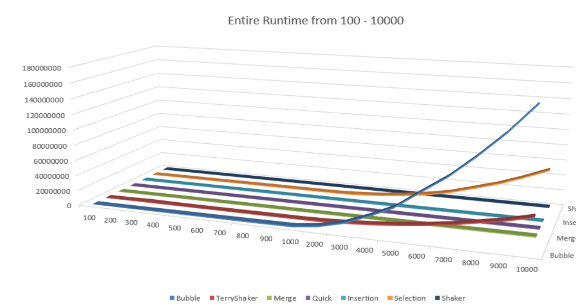
These were the averages of the test results for

$$n <= 100$$

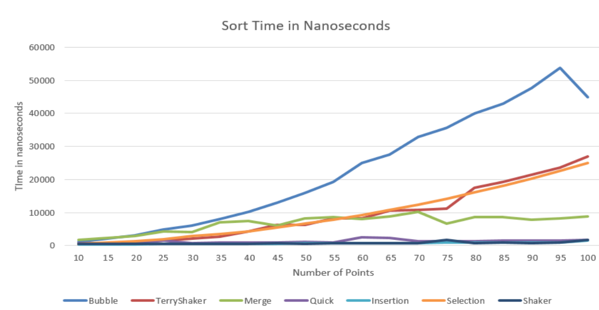
based on the random data from the original file:



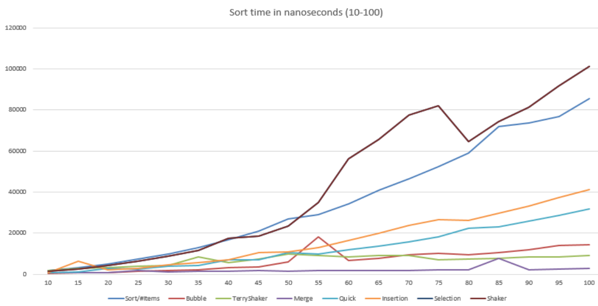
The overall graph of the 7 sort routines that we tested is as follows:



We also ran test cases for the same test sizes of n using data that was already sorted. This graph shows the results of this test for best-case scenario:



In addition, we ran the same tests for data that was in perfect reverse order. This graph shows the results of a worst-case scenario:



1.5 Run-time

For this sorting task analysis, we were required to include 5 well know sorts: Bubble, Merge, Insertion, Selection, and Quick. We were also required to select a sixth sorting algorithm of our choice. Our team chose the Cocktail shaker sort, which derives it's name from a visualization of the way the sort "bubbles" the largest item to the right, then the smallest item to the left, then the next largest item to its place on the right, etc... until all items are sorted. Our perception of the Cocktail Shaker sort was that it was an improvement over the Bubble sort because it correctly identified and placed the largest and smallest two items in their proper place on each pass. Further examination proved our initial perception to be incorrect. By definition, the Cocktail Shaker sort is indeed a Bubble sort that makes two passes: the first pass, from left to right, bubbles the largest item to the right until it is in it's proper place; the second pass, from right to left, bubbles the smallest item to the left. This alternation of directions can provide better performance in some cases. Our Bubble sort had a calculated run-time of

$$8n^2 + 5n + 2$$

and the Cocktail Shaker sort had a run-time of

$$8n^2 + 3n$$

While both the Bubble and Cocktail Shaker sorts have a Big-O of

$$n^2$$

the data shows that the Cocktail Shaker out-performs the Bubble sort. Given the misconception of the algorithm for the Cocktail Shaker sort, our team decided to try to implement what we thought was a commonsense improvement on the Bubble sort. We decided to call our sort the "Force Push" sort, given the way that each pass forces an item left and an item right. If a Jedi were standing in the middle of the area to be sorted, he would "Push" both items to either end and have them land in their appropriate location. So we created, from scratch, an algorithm to make a pass through the data and locate, but not swap, the largest and smallest items at the same time. At the end of the first iteration, we would swap the smallest element to the leftmost side, and swap the largest element to the rightmost side. The next iteration would then not need to include the first or last element when searching for the remaining minimum and maximum value and repeating the process. In our Force Push sort, the first iteration would take n passes, the second iteration would require

$$n - 2$$

passes. Each iteration would decrease n by 2. Thus, the number of iterations would be

$$n + (n-2) + (n-4) + (n-6) ... + (n-(n-2))$$

with only two element swaps per iteration. A Bubble sort would do n passes through the data on the first pass and n passes through the data on each successive iteration. Bubble sort can be improved to skip the area already sorted, thus decreasing the number of passes through the list of elements by 1 on each iteration.

This means that a better coded Bubble sort would take

$$n+(n-1)+(n-2)+(n-3)+...+(n-(n-1))$$

passes, with an average of n/2 swaps on the first iteration and

$$\frac{n-1}{2}$$

swaps on the second iteration, and so on.

The Force Push sort is also in the Big-O of:

$$n^2$$

Our Force Push sort has a run-time formula of

$$8n^2+11n+18$$

1.6 Conclusion

We found many different ideas and methods when analyzing different sort methods. Most of them are fascinating and many are ingenious. Data will always need to be sorted. We found that coding a simple sort routing can be quick and easy, as in the case of the Bubble sort. We did Bubble sort from memory and typed it straight into our class. We copied the Quick sort from the textbook and easily adapted it to our data type. Other sorts were copied straight from the Internet and adapted. But coding our Force Push sort required more testing and troubleshooting than anticipated. Our first attempt at coding the Force Push sort used recursion. This proved to be memory intensive and caused overflow errors. After the testing was done on the method's algorithm, we easily converted the implementation to an iterative approach. As can be seen from the data, we bested the Bubble sort, even though we were still in the family of

$$n^2$$

Java has a built-in sort, and we wanted to implement a TimSort from the Internet. But the ability to sort non-primitive data types using generics was beyond the scope of our knowledge. Implementing different sorts and having to adjust the coding to handle our Point data type for sorting whets the appetite to learn generics. The library of sorting methods that we have tested is already a valuable reference.

Acknowledgments

The authors of this document would like to thank Phd. C. Servin of EPCC and the LaTeXcommunity for making documentation and presets readily available to us during this paper.

Appendix A
Code

At the end of this document we provide our Java source code for the review of those grading this paper and for those interested in running future tests with Superior hardware, give Moore's Law, despite it not being a law, holds true.

Appendix B
Data

On the next page we give the table of data for our project. The data was made by summing the time of iteration for each number of elements then dividing the sum by the number of the iterations thus getting a more precise measurement. This was necessary in light of the unreliability of java run-time at the scale of nanoseconds.

Best Case																							
Sort/#Items	2	2	3	4	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
Bubble	7888	3536	682	578	666	1200	2052	3048	4841	6125	8023	10296	12971	15979	19237	25083	27610	32841	35695	40019	42996	47713	53784
TerryShaker	2373	121	294	181	197	469	723	1099	1598	2156	2830	4254	6337	6272	8272	8330	10527	10811	11227	17562	19261	21552	23622
Merge	11550	550	2422	1375	952	1721	2394	2893	4302	4158	7114	7396	6151	8228	8623	8150	8832	10260	6725	8712	8681	7915	8332
Quick	6595	371	378	1668	954	728	768	838	1503	834	907	1016	1053	1121	1051	2593	2300	1269	1318	1437	1498	1588	1571
Insertion	2945	345	316	325	346	374	422	438	522	544	560	591	634	680	726	751	795	859	883	907	973	1035	1045
Selection	2211	291	327	341	423	638	1028	1414	1995	3001	3467	4401	5437	6599	7931	9227	10804	12434	14235	16166	18170	20298	22667
Shaker	14275	477	406	348	330	550	518	512	586	599	557	583	671	646	716	741	824	820	1845	806	884	853	920
Normal Case																							
Sort/#Items	2	2	3	4	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
Bubble	9653	3561	710	817	882	3181	5182	8293	8731	9333	12321	13915	17889	20338	26595	30380	35246	42198	52757	56381	61561	65378	72846
TerryShaker	2962	142	294	192	221	484	812	1223	1786	2298	3855	5456	4783	7499	7366	9561	9546	11615	10681	12059	13089	14716	32061
Merge	10256	828	993	888	950	1997	2710	3412	4738	4884	9144	6729	8452	10038	10561	10100	11866	9284	9581	12910	11298	11774	12519
Quick	8022	482	749	785	561	1485	1477	1614	3031	2735	6050	4561	4888	5967	4990	7518	7399	6561	7880	7861	7468	7934	8820
Insertion	4226	338	337	342	382	577	899	1260	1509	3116	5260	3284	4076	4690	5745	7844	7564	9128	12515	13112	16353	17955	19179
Selection	11301	651	375	399	412	781	1292	1919	2631	4223	4486	5752	7207	8532	10098	11698	13478	15455	17535	19965	22287	24697	27299
Shaker	14158	1580	454	396	396	1039	1996	3245	4169	8309	7414	8451	11231	14144	16254	18833	20186	24662	27570	30932	38331	42033	48498
Worst Case																							
Sort/#Items	2	2	3	4	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
Bubble	12474	3966	791	651	738	1792	3186	4927	7488	9940	13093	16807	21124	27044	28865	34436	40894	46454	52271	59076	71788	73562	76711
TerryShaker	4029	115	302	210	216	483	631	885	1294	1695	2185	3202	3432	6124	18116	6825	7591	9566	10264	9446	10576	11768	14086
Merge	12299	3176	2867	1412	921	1724	3001	3071	4081	4373	8532	5816	7442	9698	8982	8297	9050	9291	6924	7522	7706	8308	8504
Quick	7820	317	357	1791	799	703	801	938	1965	1181	1326	1475	1736	1630	1680	1816	1919	1918	2200	2260	7853	2229	2655
Insertion	3859	280	309	302	334	643	1070	2692	2425	3851	4286	6961	6960	10381	9910	11801	13696	15779	18091	22285	23135	25993	28586
Selection	19454	772	1244	336	383	728	6241	2336	3019	4588	5520	7146	10390	10993	12986	16561	20122	23654	26522	26277	29750	33322	37261
Shaker	9965	402	443	459	509	1299	2568	4185	6369	8675	11464	17546	18729	23322	35045	56385	65759	77632	82202	64740	74245	81261	91966

Best Case																				
Sort/#Items	100	200	300	400	500	600	700	800	900	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	100000
Bubble	44876	80434	150549	270085	412412	468103	587071	726230	938514	1152666	4846125	12216381	24050722	38064020	59924949	81101813	107181721	135898320	169374778	24374746765
TerryShaker	26967	103599	633737	258645	133388	168859	222294	281761	352600	364782	1022020	2441659	4480667	7137701	10706078	15217550	20969188	26831359	34400197	4598257738
Merge	8889	17968	19257	24117	32188	38500	42329	48639	53360	59215	134235	196776	261918	324837	386271	452262	514678	578881	604636	7574936
Quick	1794	3480	4492	6103	7124	8833	10523	12462	13271	14551	31620	53717	68158	90012	119138	138710	159011	183929	202486	2871030
Insertion	1544	2013	2869	3700	4634	5360	6344	7140	7809	8813	35089	27769	35861	45489	54551	63444	74056	88587	92172	791386
Selection	25054	98379	107639	101283	138492	198597	248793	189375	237560	295618	1376417	3307384	6444787	10776357	16712775	25356343	34232228	45314598	57713417	7922650452
Shaker	1680	2726	3495	4380	4960	5432	6135	7056	7621	8180	43423	39754	47238	56137	57053	63186	73777	79811	87980	843497
Normal Case																				
Sort/#Items	100	200	300	400	500	600	700	800	900	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	100000
Bubble	67836	132090	270708	467566	619410	739652	911403	1147602	1402118	1749746	6790905	16368326	30954613	50319762	77063822	114843517	156929726	212750171	268921133	36847070839
TerryShaker	17632	64151	138373	199081	134453	168204	223200	279462	357966	438275	1118964	2274466	4061354	6079009	8904704	12756820	17650428	23092578	28994338	4475158436
Merge	9156	20348	29999	42730	52718	69158	76159	87676	99249	111110	240483	375783	511679	657137	799529	926532	1026412	1185127	1337457	18484876
Quick	8788	21100	29305	32333	32352	38623	45129	53417	58907	68156	150975	238855	318080	417009	503465	598058	644208	730609	838092	11124348
Insertion	21319	67797	138949	134314	104304	146871	201899	261992	292904	221830	881969	2058399	3576587	5695812	8254331	11483350	15448923	20738044	26133028	4095861964
Selection	30807	111756	149997	156626	209950	290081	323522	341934	431376	531914	2077791	4392078	7708285	12224421	19459640	28266981	39589021	51663564	65628800	9780080884
Shaker	53979	171267	203583	251622	332905	501284	617706	620544	1084835	934204	3609678	8268109	15340345	24137649	36910572	54380420	74905477	100584675	129920280	22205007789
Worst Case																				
Sort/#Items	100	200	300	400	500	600	700	800	900	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	100000
Bubble	85391	171805	176761	293651	421337	576527	775037	980233	1234708	1520051	6376022	16165078	32710965	52264909	76319578	105737268	138071732	175956884	216365986	24632947675
TerryShaker	14543	57291	131074	243702	349406	283580	85484	108310	137765	165982	735827	1938847	4004012	6924693	10955978	16152963	21486298	28218039	36095262	4196718249
Merge	9193	17015	20030	24121	30594	39581	45352	50699	54907	61550	139417	223686	300977	374071	432139	459974	507117	544600	604070	7894107
Quick	2796	6095	8779	11409	14318	17126	20406	21827	25818	27671	60569	90780	121434	155046	183961	221307	251366	289101	321612	4318745
Insertion	31787	132064	287823	222405	118670	167516	225680	294517	368808	464200	1895683	4759899	9168014	15329334	23361387	32714332	44473362	56998013	72056910	8133531082
Selection	41261	172596	382807	391345	543245	788830	1067651	1400581	1744736	2187463	8657918	19731847	34897514	54824540	79013998	108939843	144050766	180242193	223699895	23032869226
Shaker	101266	365033	163966	258873	395758	596612	765400	992572	1277731	1617780	6524235	15030252	27920556	45892034	67128551	95117178	125434959	162776076	203185637	23419809756

Source Code

by Jason Ivey & Terry Speicher using: LaTeX & Dr Java.

October 2016

1 Goal

To read the 100kpoint.txt file and sort the data within it providing reliable data on the run-time of each algorithm.

```java
  class Point{
    private double x;
    private double y;
    public Point(double x, double y){
      this.x = x;
      this.y = y;
    }
    public double getX(){
      return this.x;
    }
    public double getY(){
      return this.y;
    }
  }
```

```java
//Terry Speicher and Jason Ivey

/**
 * Collection of sorting algorithms to be compared, along with a few utilitarian methods
 * @author Terry Speicher
 * @author Jason Ivey
 */
public class SortAlgorithms{

  /**
   * Constructor doesn't have to do anything
   */
  public SortAlgorithms(){

  }

  /**
   * Not a complete comparison of all items to see if they are in order, but quickly checks
   * consecutive items to make sure that the smaller item is first.
   * @param data Point[] of some size to be processed
   * @return boolean Return true if each consecutive item is less than the one after it.
   */
  public boolean isSorted(Point [] data){

    boolean sortedYN = true;
    int i = 0;
    while (i < data.length - 2 && sortedYN == true) {
      if (data[i].getX() > data[i+1].getX())
        sortedYN = false;
      i++;

    }
    return sortedYN;
  }  //end isSortedYN


  /**
   * Compare the first 100 items in each given Point array to see if they are equal.  Used
   * as a testing routine to work out some bugs when learning how to pass the data by
   * reference.
   * @param data Point[] of some size to be processed
   * @param test Point[] of some size to be processed
```

```
   * @return boolean Return true if the first 100 x coordinates are the same in the
   * two arrays
   */
  public boolean looksEqual(Point [] data, Point [] test){

    boolean lookSame = true;
    for (int i = 0 ; i < 100 ; i++)
      if (data[i].getX() != test[i].getX())
      lookSame = false;
    return lookSame;
  } //end looksEqual

//  ***********  Begin BubbleSort *****************

  /**
   * BubbleSort was coded by the authors from scratch.  This is the worst case sort because
   * it does not even adjust the inner loop to start at the first nonsorted element - it
   * always starts at the beginning.
   * @param d Point[] of some size to be processed
   */
  public void bubbleSort(Point [] d){
    for(int x = 0; x < d.length; x++){
      for(int y = 0; y < d.length-1; y++){
        if ( d[y].getX() > d[y+1].getX() ) {
          Point temp = d[y];
          d[y] = d[y+1];
          d[y+1] = temp;
        }
      }
    }
  }

//************* End Bubble Sort *******************

//  *************  Begin QuickSort Methods  ******************

  /**
   * Copied from the textbook.  Improved on by adding a check in the partitioning
   * routine to check "if (endOfLeftList != scan)" before swapping, because we found
   * that the swap routine was sometimes being called to swap an element with itself if
   * the "endOfLeftList" equaled "scan".
   * @param data Point[] of some size to be processed
   */
  public void quickSort(Point [] data){
    doQuickSort(data,0,data.length-1);


  }  // end quickSort

  /**
   * sub method to start recursion process.  Required to
   * pass in the two indexes for processing the sub array.
   * Initially, we pass in the absolute beginning and
   * absolute ending elements.
   * @param array Point[] to be sorted
   * @param start int that shows where the start of the subarray to be sorted is
   * @param end int that shows where the end of the bubarray to be sorted is
   **/
  private void doQuickSort(Point[] array, int start, int end){
    int pivotPoint;
    if (start < end){
      pivotPoint = partition(array, start, end);
      doQuickSort(array,start,pivotPoint-1);
      doQuickSort(array,pivotPoint+1,end);
    }
  } // end doQuickSort

  /**
   * Partition the given array from array[start] to array[end] and put
   * pivot element in middle.  Then move all elements with values less
   * than the pivot point to the left of the pivot point and move
   * all elements with values greater than the pivot point to the right
   * of the pivot point.
   * @param array Point[] to be sorted
   * @param start int that shows where the start of the subarray to be partitioned is
   * @param end int that shows where the end of the bubarray to be partitioned is
   **/
  private int partition(Point[] array, int start, int end){

    double pivotValue;
    int endOfLeftList;
```

```java
    int mid;

    mid = (start + end) / 2;

    //take the first element of the array and swap
    //it with the middle element.  I don't know why,
    //except to keep with the idea of using the middle
    //element as the pivot
    swap(array,start,mid);

    pivotValue = array[start].getX();
    endOfLeftList = start;
    for (int scan = start + 1; scan <= end; scan++){
      if (array[scan].getX() < pivotValue){
        endOfLeftList++;
        if (endOfLeftList != scan)
          swap(array,endOfLeftList,scan);
      }
    }//end of for

    swap(array,start,endOfLeftList);

    return endOfLeftList;

  }//end partition

  /**
   * Swap elements array[a] and array[b]
   * @param array Point[] with elements to be swapped
   * @param a index of an element to be swapped
   * @param b index of other element to be swapped
   **/
  private void swap(Point[] array, int a, int b){

    Point temp = array[a];
    array[a] = array[b];
    array[b] = temp;

  }//end swap

// **************  End QuickSort Methods   *****************

//  *************  Begin MergeSort Method  ******************

  /*************************************************************************************
    *    With online help from:
    *    Title: howotodoinjava.com
    *    Author: Lokesh Gupta
    *    Date: October 23, 2015
    *    Availability: http://howtodoinjava.com/algorithm/merge-sort-java-example/
    *
    *************************************************************************************/
  /**
   * Merge Sort method to sort an array of data points
   * @param data Point[] of some size to be processed
   */
  public void mergeSort(Point[] data){

    if (data.length <= 1){
      return;
    }

    Point[ ] half1 = new Point[ data.length / 2 ];
    Point[ ] half2 = new Point[ data.length - half1.length ];

    System.arraycopy(data, 0, half1, 0, half1.length);
    System.arraycopy(data, half1.length, half2, 0, half2.length);

    mergeSort(half1);
    mergeSort(half2);

    merge(half1, half2, data);
    return;

    /**
     * Main recursive method to merge sorting.
     * divides Points array into smaller pieces.
     * Makes use of merge(Point[],Point[],Point[]) to combine and sort data
     **/
  }
  /**
```

```java
   * Merge elements of array[a] and array[b] into result
   * @param half1 Point[] half to be combined with brother in order
   * @param half2 Point[] brother of half1
   * @param result Point[] to be returned when brothers combined and sorted.
   **/
  private static void merge(Point[] half1, Point[] half2, Point[] result){
    int x = 0;
    int y = 0;
    int merge = 0;

    while(x < half1.length && y < half2.length ){
      if(half1[x].getX() < half2[y].getX()){
        result[merge] = half1[x];
        x++;
      }
      else{
        result[merge] = half2[y];
        y++;
      }
      merge++;
    }
    System.arraycopy(half1, x, result, merge, half1.length - x);
    System.arraycopy(half2, y, result, merge, half2.length - y);
  }
  //  *************  End mergeSort Methods  *****************

  //  *************  Begin insertionSort Methods  *****************

  /*************************************************************************************
    *    With online help from:
    *    Title: http://www.java2novice.com/
    *    Author: N/A
    *    Date: N/A
    *    Availability: http://www.java2novice.com/java-interview-programs/insertion-sort/
    *************************************************************************************/


  /**
   * Sort Point[] by inserting unsorted points into the correct positions in sorted Point[]
   * @param array Point[] array to be sorted
   **/

  public static void insertionSort(Point array[]) {
    int n = array.length; // Limiter initalized to length of Point[] array
    for (int j = 1; j < n; j++) {
      Point key = array[j]; // Key to comparison
      int i = j-1; // Interloop counter, one less than 'j'
      while ( (i > -1) && ( array[i].getX() > key.getX() ) ) {
        array [i+1] = array [i];
        i--;
      }
      array[i+1] = key; // Inserting key into sorted portion of array.
    }
  }

  //  *************  End insertionSort Methods  *****************


  //  *************  Begin selectionSort Methods  *****************

  /*************************************************************************************
    *    With online help from:
    *    Title: Sorting.java
    *    Author: Lewis/Loftus
    *    Date: N/A
    *    Availability: http://www.ics.uci.edu/~stasio/winter06/Lectures/Lec7code/ComparableExample/Sorting.java
    *************************************************************************************/

  /**
   * Sorts the Point array of objects using the selection sort algorithm.
   * @param input Point[] array to be sorted
   **/

  public static void selectionSort (Point[] input)
  {
    int min; // Min value
    Point temp; // Temp storage of Points

    for (int index = 0; index < input.length-1; index++)
    {
```

```
      min = index; // Initializing min
      for (int scan = index+1; scan < input.length; scan++)
        if (input[scan].getX() < input[min].getX())
        min = scan; // update min

      // Swap the values
      temp = input[min];
      input[min] = input[index];
      input[index] = temp;
    }
  }


//  **************   Cocktail Shaker Sort   ********************
//http://www.javacodex.com/Sorting/Cocktail-Sort

  /**
   * Cocktail Shaker Sort
   * @param array Point[] of some size to be processed
   */
  public static void cocktailShakerSort( Point[] array ){
    boolean swapped;
    do {
      swapped = false;
      for (int i =0; i<=  array.length  - 2;i++) {
        if (array[ i ].getX() > array[ i + 1 ].getX()) {
          //test whether the two elements are in the wrong order
          Point temp = array[i];
          array[i] = array[i+1];
          array[i+1]=temp;
          swapped = true;
        }
      }
      if (!swapped) {
        //we can exit the outer loop here if no swaps occurred.
        break;
      }
      swapped = false;
      for (int i= array.length - 2;i>=0;i--) {
        if (array[ i ].getX() > array[ i + 1 ].getX()) {
          Point temp = array[i];
          array[i] = array[i+1];
          array[i+1]=temp;
          swapped = true;
        }
      }
      //if no elements have been swapped, then the list is sorted
    } while (swapped);
  }

  /**
   * Force Push Sort was coded by Terry Speicher to show how he *thought* the Cocktail
   * Shaker was supposed to work.  It is kind of a "double ended" selection sort.
   * @param p Point[] of some size to be processed
   */
  public void forcePush(Point [] p){

    Point temp = new Point(0.0,0.0);

    Point swapper = new Point(0.0,0.0);

    for (int start = 0, end = p.length -1 ; start < end  ; start++, end--){
      if (start != end) { //only 1 element.  We are done.
        if ( (end - start) == 1) {
          //only two elements left
          //compare and swap if necessary
          if (p[start].getX() > p[end].getX()){
            temp = p[start];
            p[start] = p[end];
            p[end] = temp;
          }
        } else {
          //more than 2 elements - go through and find max and min and swap
          double maxValue;
          double minValue;
          int minValueIndex;
          int maxValueIndex;

          minValue = maxValue = p[start].getX();  //set min and max to first element
          minValueIndex = maxValueIndex = start;

```

```java
          for (int i = start + 1 ; i <= end ; i++){  //go through array to find max and mins
            if (p[i].getX() < minValue) {
              minValue = p[i].getX();
              minValueIndex = i;
            } else
              if (p[i].getX() > maxValue) {
              maxValue = p[i].getX();
              maxValueIndex = i;
            }
          }  //after this for statement, we know the locations of the max and min elements

          swapper = p[start];
          p[start] = p[minValueIndex];
          p[minValueIndex] = swapper;

          //this handles the funny case where the max value was in the start position
          //but then we moved it when we swapped it with the element that actually should
          //end up in the start position.  That was the swap above, which would leave our
          //max element in the position with the index of 'minValueIndex'.
          if (maxValueIndex == start)
            maxValueIndex = minValueIndex;

          swapper = p[end];
          p[end] = p[maxValueIndex];
          p[maxValueIndex] = swapper;


        }

      }
    }//end main for loop
  } //end forcePush

  /**
   * Utilitarian method to print out the x value from an array of Points.  This was used
   * to find the odd case in the Force Push sort where the largest element in the portion
   * of the array to be sorted was in the first slot and therefore got moved when the
   * smallest element was swapped into its place.  The method was left in the class for
   * documentation purposes only.
   * @param a Point[] of some size to be processed
   */
  public void printArray(Point [] a){
    for (int i = 0 ; i < a.length ; i++) {
      System.out.printf("%.2f ",a[i].getX());
    }
    System.out.println();
  }


}
```

```java
//Terry Speicher and Jason Ivey

/**
 * Read file into an array and present different sized sub arrays of those points
 * to each different sort routines and record timed results
 * @author Terry Speicher
 * @author Jason Ivey
 */
import java.util.*;
import java.util.Arrays;
import java.io.*;

public class SortAlgorithmsTester{

  private static Point[] data = new Point[100000];

  /**
   * Main body of Sort Tester.
   * @param args standard header String[]
   */
  public static void main(String args[]){

    /** Create int array representing the number of elements that will be taken from the
      * beginning of the array of random items
      */
    int [] testCases = {2,2,3,4,5,10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,85,90,95,100,
      200,300,400,500,600,700,800,900,1000,2000,3000,4000,5000,6000,7000,
      8000,9000,10000,data.length};

    // Set the number of times the data will be tested and averaged.
```

```java
      int iterations = 10;
      //results table is 9 by however many testCases there are
      long [][] resultsTable = new long[8][testCases.length];
      boolean testing = false; //Turn on/off verbose intermediate findings

      read(); //one time read of data[]

      SortAlgorithms sort = new SortAlgorithms();  //create class of sort methods

      //Main counter for determined number of test cases
      for (int counter = 1 ; counter <= iterations ; counter++){

        //Visual output of loop # currently being processed
        System.out.println("Iteration#" + counter + " of " + iterations);

        //Print headings
        if (testing) System.out.print("Sort/#Items,");
        for (int i = 0 ; i < testCases.length ; i++)
          if (testing) System.out.print(testCases[i] + ",");
        if (testing) System.out.println();

        //Print each sort name and test results from each test of n elements
        if (testing) System.out.print("BubbleSort,");
        for (int i = 0 ; i < testCases.length ; i++){
          long startTime = System.nanoTime();
          sort.bubbleSort(Arrays.copyOfRange(data,0,testCases[i]));
          long estimatedTime = System.nanoTime() - startTime;
          if (testing) System.out.print(estimatedTime + ",");
          resultsTable[1][i] += estimatedTime;
        }
        if (testing) System.out.println();

        //Keep printing name of sort and results
        if (testing) System.out.print("ForcePush,");
        for (int i = 0 ; i < testCases.length ; i++){
          long startTime = System.nanoTime();
          sort.forcePush(Arrays.copyOfRange(data,0,testCases[i]));
          long estimatedTime = System.nanoTime() - startTime;
          if (testing) System.out.print(estimatedTime + ",");
          resultsTable[2][i] +=estimatedTime;
        }
        if (testing) System.out.println();

        if (testing) System.out.print("MergeSort,");
        for (int i = 0 ; i < testCases.length ; i++){
          long startTime = System.nanoTime();
          sort.mergeSort(Arrays.copyOfRange(data,0,testCases[i]));
          long estimatedTime = System.nanoTime() - startTime;
          if (testing) System.out.print(estimatedTime + ",");
          resultsTable[3][i] +=estimatedTime;
        }
        if (testing) System.out.println();

        if (testing) System.out.print("QuickSort,");
        for (int i = 0 ; i < testCases.length ; i++){
          long startTime = System.nanoTime();
          sort.quickSort(Arrays.copyOfRange(data,0,testCases[i]));
          long estimatedTime = System.nanoTime() - startTime;
          if (testing) System.out.print(estimatedTime + ",");
          resultsTable[4][i] +=estimatedTime;
        }
        if (testing) System.out.println();

        if (testing) System.out.print("InsertionSort,");
        for (int i = 0 ; i < testCases.length ; i++){
          long startTime = System.nanoTime();
          sort.insertionSort(Arrays.copyOfRange(data,0,testCases[i]));
          long estimatedTime = System.nanoTime() - startTime;
          if (testing) System.out.print(estimatedTime + ",");
          resultsTable[5][i] +=estimatedTime;
        }
        if (testing) System.out.println();

        if (testing) System.out.print("SelectionSort,");
        for (int i = 0 ; i < testCases.length ; i++){
          long startTime = System.nanoTime();
          sort.selectionSort(Arrays.copyOfRange(data,0,testCases[i]));
          long estimatedTime = System.nanoTime() - startTime;
          if (testing) System.out.print(estimatedTime + ",");
          resultsTable[6][i] +=estimatedTime;
        }
```

```java
      if (testing) System.out.println();

      if (testing) System.out.print("ShakerSort,");
      for (int i = 0 ; i < testCases.length ; i++){
        long startTime = System.nanoTime();
        sort.cocktailShakerSort(Arrays.copyOfRange(data,0,testCases[i]));
        long estimatedTime = System.nanoTime() - startTime;
        if (testing) System.out.print(estimatedTime + ",");
        resultsTable[7][i] +=estimatedTime;
      }
      if (testing) System.out.println();

    }

    //In verbose mode, print out time totals table
    if (testing) {
      System.out.println("------------------------Totals over " + iterations + " iterations
          ----------------------------");     System.out.print("Sort/#Items,");
      for (int i = 0 ; i < testCases.length ; i++)
        System.out.print(testCases[i] + ",");
      System.out.println();

      for (int a = 1; a <=7 ; a++){
        switch (a) {
          case 1:
            System.out.print("Bubble,");
            break;
          case 2:
            System.out.print("ForcePush,");
            break;
          case 3:
            System.out.print("Merge,");
            break;
          case 4:
            System.out.print("Quick,");
            break;
          case 5:
            System.out.print("Insertion,");
            break;
          case 6:
            System.out.print("Selection,");
            break;
          case 7:
            System.out.print("Shaker,");
            break;
        }  //end switch case

        for (int j = 0 ; j < resultsTable[0].length ; j++){
          System.out.print(resultsTable[a][j] + ",");
        }
        System.out.println();


      }
    }

    //Same loop as above, but print out the averages instead of the total time
    System.out.println("------------------------Averages over " + iterations + " iterations
        ----------------------------");

    System.out.print("Sort/#Items,");
    for (int i = 0 ; i < testCases.length ; i++)
      System.out.print(testCases[i] + ",");
    System.out.println();

    for (int a = 1; a <=7 ; a++){
      switch (a) {
        case 1:
          System.out.print("Bubble,");
          break;
        case 2:
          System.out.print("ForcePush,");
          break;
        case 3:
          System.out.print("Merge,");
          break;
        case 4:
          System.out.print("Quick,");
          break;
        case 5:
          System.out.print("Insertion,");
```

```
          break;
        case 6:
          System.out.print("Selection ,");
          break;
        case 7:
          System.out.print("Shaker ,");
          break;
      } //end switch case

      for (int j = 0 ; j < resultsTable[0].length ; j++){
        System.out.print(resultsTable[a][j]/ iterations+ ",");
      }
      System.out.println();


    }

  }

  /**
   * Read in the file with the pairs of point (x,y) coordinates and place in the data[]
   *
   */
  private static void read(){

    try {
      File file = new File("100000Points.txt");
      Scanner scan = new Scanner(file);

      int x = 0;
      String line;
      String token[];
      while(scan.hasNext()){
        line = scan.nextLine();
        token = line.split("\t");
        data[x] = new Point(Double.parseDouble(token[0]), Double.parseDouble(token[1]));
        x++;
      }

      //confirm number of elements read
      System.out.println("We have " + x + " points");
      scan.close();
    }
    catch (Exception e){
      System.out.println("Error in reading file");
    }
  }


}
```