Experimental identification of sorting algorithms

Abstract

Identifying the sorting algorithms through empirical means of observation and experimenting with various methods to obtain the identity of the sorting algorithm being used in the SortTrials.class. The five sorting methods that are being identified are mergesort, quicksort with no randomization, randomized quicksort, insertion sort, or selection sort. The various methods that one can use to obtain the identity of each sorting method includes timing the method from invocation to the return of the place where the method was called in this case it is SortTrialsClient.class that is invoking the sorting methods of SortTrials.class. After the timing is done there are other sub identification methods that can be used which are comparing from known times of various sorting algorithms. The second method is to look at the list as it is modified while it is being sorted, since each method has a unique signature to how it sorts one can distinguish each sort correctly.

Introduction

The experiment of identifying the sorting algorithms being implemented in the five public methods of the SortTrials.java are mergesort, quicksort with no randomization, randomized quicksort, insertion sort, and selection sort. The objective is to correctly identify the five sorts based off empirical methods. The importance of this experiment is to show that an empirical approach can be applied to correctly identify sorting methods based off experimental results made. To determine the different types of sorting methods one can look at the log Ratio to determine the time complexity of the algorithm in terms of big-Oh, which then can be compared to known big-Oh values and process of elimination one can come to an conclusion that the method is that sort. If one can not rule out other choices by using looking at known types then eliminating some choices can be achieved. . In order to achieve this timing of each method must be done and the ratio must be calculated. Other approaches to narrow down the identification process in the empirical approach can be done by observing the modifications of the list that is being sorting by a single sort method that is being analyzed to determine the identity of it.

Procedures

In this experiment I started out by not writing any new code at first and realized that using words in each element is not needed. I then chose a word that would be spelled correctly when in alphabetical order and that each character of the word would be a separate element to verify by looking that it was in fact in natural order. The word chosen was aegilops , which is goat grass. I ran the program in canvas in JGrasp and dragged the List that was sent to be sorted through and watched as that list changed and wrote down each change that occurred. By writing each modification of the list I was able to distinguish what sort was which to verify or isolate findings that would come later by timing the invocation of the sorting methods. After observing the modifications that took place to the list and recording each change I marked where each change took place within each list. To incorporate timing into the empirical process to find what sorting method was which. I had to figure out a way to run the experiment a number of times without having to change the problem size each time manually. The instructor provided a simple example of a client to use, which would require changing the problem size each time, which was the opposite of what was need to be done. I looked at the power point slide 4 of " Algorithm_Analysis.pptx.pdf" which gave an example of what might be expected as an output. I also wanted to save time by having a switch statement that would change what sorting method was to be invoked to have the data all in one output. I managed to accomplish this by have a loop out side of the previous loop that would iterate and change the variable that would switch what case to be called. I included in the output a way to tell which sorting case was being invoked. In order to increase the size of the problem size I wrote another method to increase the size by using an loop to iterate a given number of times and store each number iterated to an element to be stored until the number of times given has been reached, this created an already sorted list of integers that was about to be used to increase the problem size. I then needed to write a loop that would execute a number of times and calculate the ratio and log ratio for each iteration of the problem size change and print the output to the console. In order for the ratio to be calculated there must be a previous run which that meant that the first iteration would not have a ratio also it was apparent in the power point slide mentioned earlier. The formula used to calculate the ratio is $Ratio = \frac{Current\ Time}{Previous\ Time}$, where current time is the present iteration time and the previous time is the last iteration's time. To calculate the log ratio I needed to use the Math class and the static methods of it in order to calculate the formula for the log ratio which is $\log Ratio = \log_2(\frac{Current\ Time}{Previous\ Time})$. After being let down that the Math class does not provide a way to obtain base two easily I resorted to use algebra to provide a way using the static methods provided to obtain the base two equivalent of the ratio. The formula used to obtain this is $\log_2 Ratio = \frac{\log(\frac{Current\ Time}{Previous\ Time})}{\log(2)}$. After writing and testing the program I notice I needed to choose a proper problem size value so the program and each sorting algorithm would output meaningful data. I chose to start with 128 and then double in size to the value of 262144 which is 12 iterations of the inner loop while doubling the N size. I also needed to write a method to populate an array for the worst case for quicksort with no randomization to reveal the identity of the method. To do this I used the same method as to populate the array with consecutive numbers in natural order, but this time to populate the array for the worst case I needed to start at the end of the array such that N is the size of the array and to get the last element to populate N -1 is used and then decrement by one and store in each element until the index is 0. By doing this I have created an reverse natural order to have to use when quicksort is found and when it is found to distinguish it from the other I use the worst case method to identify it. I also had to comment out the shuffling before to allow the worst case array stay the worst case array. I also used another switch statement to increase the problem size for sorts 3, 4, and 5 since during debugging they were sorting to fast to get meaningful data. So I started with 4096 and doubling its size until 8388608 was reached, by doing so the data that was returned was meaningful.

Results and Discussion

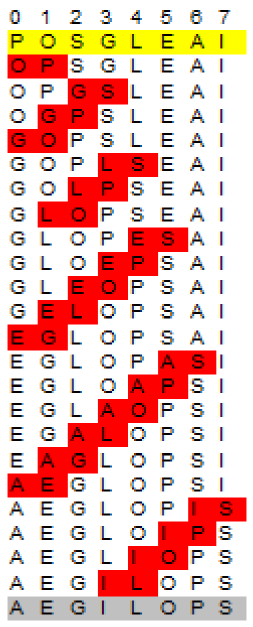
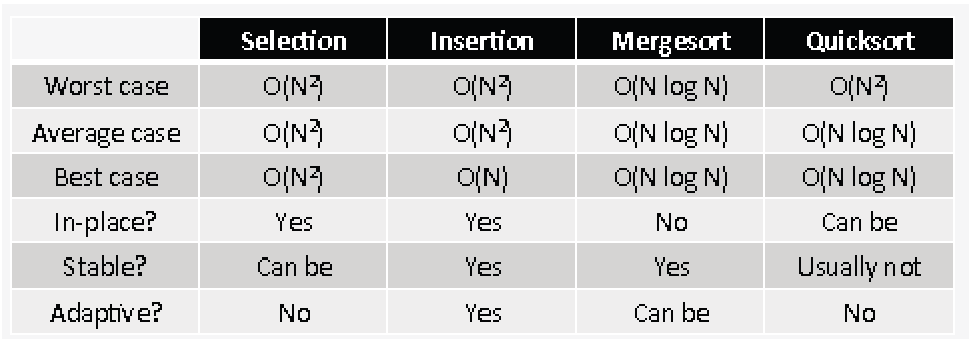


Figure 1. Visual representation of sort1() given an array of Strings {P,O,S,G,L.E.A.I} Where the first row is identifies the element number, the second row, highlighted yellow, identifies the starting array, the last row identifies the array in natural order, highlighted gray, the red highlights indicate the modifications of the array.

```
--------------------------[Sort #1 Time Trial]------------------------
Run       N         T(N)           Ratio            lg Ratio          Sorted
----------------------------------------------------------------------
0         128       .002
1         256       .004           2.000            1.000             true
2         512       .012           3.000            1.585             true
3         1024      .006           .500             -1.000            true
4         2048      .003           .500             -1.000            true
5         4096      .019           6.333            2.663             true
6         8192      .074           3.895            1.962             true
7         16384     .295           3.986            1.995             true
8         32768     1.188          4.027            2.010             true
9         65536     4.823          4.060            2.021             true
10        131072    20.634         4.278            2.097             true
11        262144    93.065         4.510            2.173             true
```

Table 1. The first segment of data that was generated by the source code to find the big-Oh of method sort1(), there are five different columns: Run, N, T(N), Ratio, lg Ratio. Each column represents essential data needed to discover the time complexity which is the big-Oh.



	Selection	Insertion	Mergesort	Quicksort
Worst case	$O(N^2)$	$O(N^2)$	O(N log N)	$O(N^2)$
Average case	$O(N^2)$	$O(N^2)$	O(N log N)	O(N log N)
Best case	$O(N^2)$	O(N)	O(N log N)	O(N log N)
In-place?	Yes	Yes	No	Can be
Stable?	Can be	Yes	Yes	Usually not
Adaptive?	No	Yes	Can be	No

Diagram 1. Snippet of slide 47 of Sorting.pptx.

To identify the sorting algorithm that is being applied by the method sort1() of SortTrials.class one can look at figure 1. In the figure the first modification or switch occurs at the beginning of the array and with this modification the elements that were switched are not in their natural order position relative to the rest of the elements in the array. By this modification alone we can rule out selection sort. The next modification is done consecutively after the other and comparing and switching with the element next to it if it is less than the other element. By this we can conclude that this sort is insertion sort. Table one shows that the insertion algorithm that was used is $O(N^2)$, this is known by observing the column “lg ratio" while it is verging to 2, by this we can say that it is $O(N^2)$. The finding is also consistent with Diagram 1.

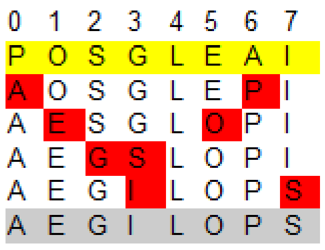
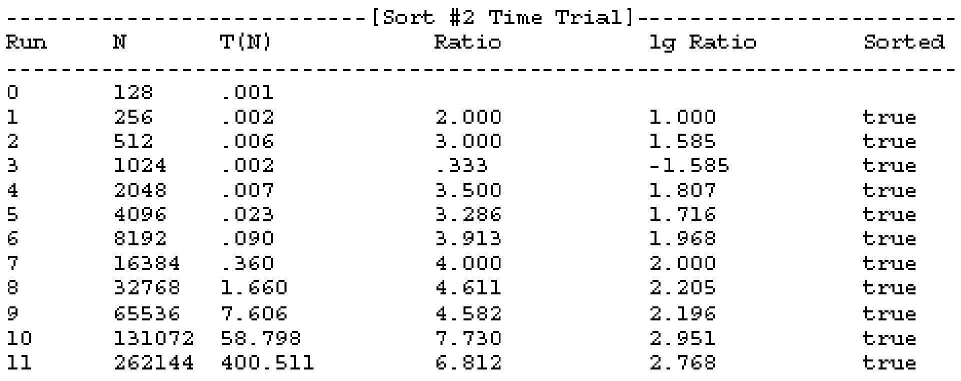


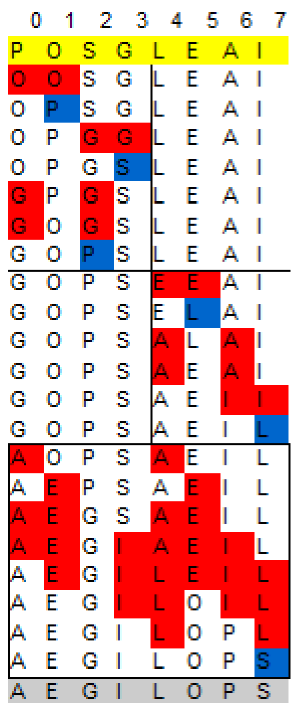
Figure 2.Visual representation of sort2() given an array of Strings {P,O,S,G,L.E.A.I}. Where the first row is identifies the element number, the second row, highlighted yellow, identifies the starting array, the last row identifies the array in natural order, highlighted gray, the red highlights indicate the modifications of the array.



```
---------------------------[Sort #2 Time Trial]------------------------
Run     N       T(N)            Ratio           lg Ratio        Sorted
----------------------------------------------------------------------
0       128     .001
1       256     .002            2.000           1.000           true
2       512     .006            3.000           1.585           true
3       1024    .002            .333            -1.585          true
4       2048    .007            3.500           1.807           true
5       4096    .023            3.286           1.716           true
6       8192    .090            3.913           1.968           true
7       16384   .360            4.000           2.000           true
8       32768   1.660           4.611           2.205           true
9       65536   7.606           4.582           2.196           true
10      131072  58.798          7.730           2.951           true
11      262144  400.511         6.812           2.768           true
```

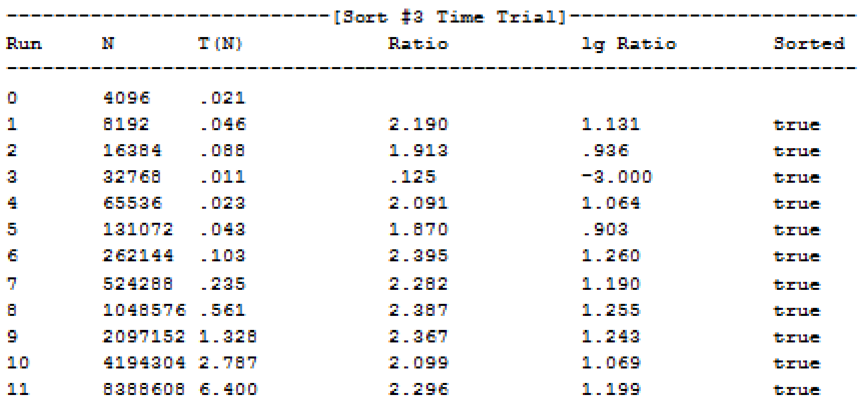
Table 2. The first segment of data that was generated by the source code to find the big-Oh of method sort2(), there are five different columns: Run, N, T(N), Ratio, lg Ratio. Each column represents essential data needed to discover the time complexity which is the big-Oh.

To identify the sorting algorithm that is being applied by the method sort2() of SortTrials.class one can look at figure 2. In the figure the first modification that is done relative to the other elements is the natural order position of the element. This is not to rule out any others just yet. The second modification that is done is also in the natural order position. As one can see this pattern is obvious throughout the array. This pattern is unique to the sorting algorithm of selection sort. In Table 2 by observing the column “lg Ratio” one can see that it is staying around 2 even though it is a slightly high two it drops two tenths before the ending. With this observation of table 2 we can say the selection sort algorithm is O(N^2) and is consistent with diagram 1.



0	1	2	3	4	5	6	7
P	O	S	G	L	E	A	I
O	O	S	G	L	E	A	I
O	P	S	G	L	E	A	I
O	P	G	G	L	E	A	I
O	P	G	S	L	E	A	I
G	P	G	S	L	E	A	I
G	O	G	S	L	E	A	I
G	O	P	S	L	E	A	I
G	O	P	S	E	E	A	I
G	O	P	S	E	L	A	I
G	O	P	S	A	L	A	I
G	O	P	S	A	E	A	I
G	O	P	S	A	E	I	I
G	O	P	S	A	E	I	L
A	O	P	S	A	E	I	L
A	E	P	S	A	E	I	L
A	E	G	S	A	E	I	L
A	E	G	I	A	E	I	L
A	E	G	I	L	E	I	L
A	E	G	I	L	O	I	L
A	E	G	I	L	O	P	L
A	E	G	I	L	O	P	S
A	E	G	I	L	O	P	S

Figure 3.Visual representation of sort3() given an array of Strings {P,O,S,G,L.E.A.I}. Where the first row is identifies the element number, the second row, highlighted yellow, identifies the starting array, the last row identifies the array in natural order, highlighted gray, the red highlights indicate the modifications of the array.



```
---------------------------[Sort #3 Time Trial]------------------------
Run     N        T(N)          Ratio           lg Ratio        Sorted
-----------------------------------------------------------------------
0       4096     .021
1       8192     .046          2.190           1.131           true
2       16384    .088          1.913           .936            true
3       32768    .011          .125            -3.000          true
4       65536    .023          2.091           1.064           true
5       131072   .043          1.870           .903            true
6       262144   .103          2.395           1.260           true
7       524288   .235          2.282           1.190           true
8       1048576  .561          2.387           1.255           true
9       2097152  1.328         2.367           1.243           true
10      4194304  2.787         2.099           1.069           true
11      8388608  6.400         2.296           1.199           true
```

Table 3.The first segment of data that was generated by the source code to find the big-Oh of method sort3(), there are five different columns: Run, N, T(N), Ratio, lg Ratio. Each column represents essential data needed to discover the time complexity which is the big-Oh.

To identify the sorting algorithm that is being applied by the method sort3() of SortTrials.class one can look at figure 3. In the figure the first modification is strange it places an duplicate of the item in the element slot to which it will swap to, while maintaining the removed value in storage to place else where. Observation of figure 3 further shows that modifications are only being done to the left side of the list, given that between element I and L is the middle. There is a black line visually separating this distinction. The second cluster of modifications are being done only to elements of the right. The last half of the figure the modifications are being done throughout the array. By this pattern one can conclude that sort3() is merge sort. Finding the time complexity of this algorithm in terms of big-Oh we can conclude that it is not a quadratic, that leaves us with O(N Log N) as the time complexity in terms of big-Oh for this algorithm, this consistent with Diagram 1.

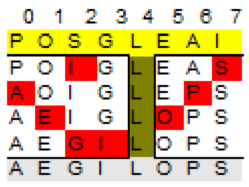
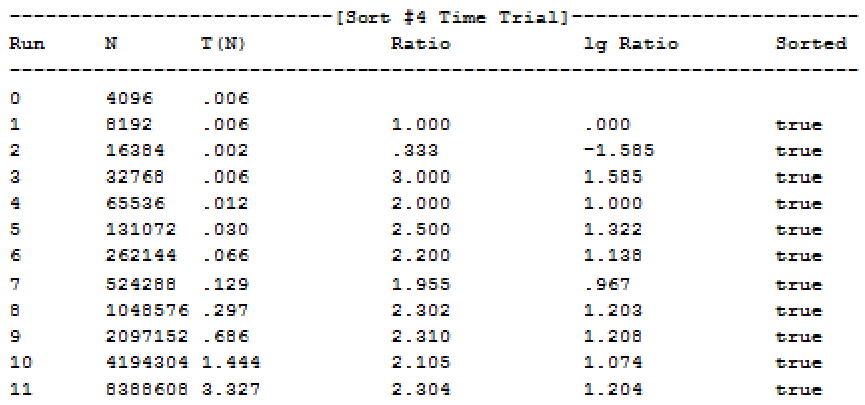


Figure 4.Visual representation of sort4() given an array of Strings {P,O,S,G,L.E.A.I}. Where the first row is identifies the element number, the second row, highlighted yellow, identifies the starting array, the last row identifies the array in natural order, highlighted gray, the red highlights indicate the modifications of the array.



```
---------------------------[Sort #4 Time Trial]------------------------
Run      N        T(N)            Ratio           lg Ratio         Sorted
-----------------------------------------------------------------------
0        4096     .006
1        8192     .006            1.000           .000             true
2        16384    .002            .333            -1.585           true
3        32768    .006            3.000           1.585            true
4        65536    .012            2.000           1.000            true
5        131072   .030            2.500           1.322            true
6        262144   .066            2.200           1.138            true
7        524288   .129            1.955           .967             true
8        1048576 .297             2.302           1.203            true
9        2097152 .686             2.310           1.208            true
10       4194304 1.444            2.105           1.074            true
11       8388608 3.327            2.304           1.204            true
```

Table 4. The first segment of data that was generated by the source code to find the big-Oh of method sort4(), there are five different columns: Run, N, T(N), Ratio, lg Ratio. Each column represents essential data needed to discover the time complexity which is the big-Oh.

To identify the sorting algorithm that is being applied by the method sort4() of SortTrials.class one can look at figure 4. The figure shows that the element does not move similar to figure 2, but unlike figure 2 the first modification of the array did not leave a natural order element in place. By leaving the element 'L' in place throughout the duration of the sort, we can think that this maybe a pivot point in the array. Give that 'L' is the pivot point and observation is made that the goal from the modifications is to place elements that are less then the pivot to the left and items greater than the pivot to the right and thus sort the array. By this we can conclude that the sorting algorithm being used is quicksort. To find out if it is quicksort with no randomization or an randomized quicksort, we have to do another test for worst case to see if sort4() is with randomization of the pivot. Figure 4(a) looks similar to selection sort if it was given the same array. Table 4(a) shows sort4() with worst case problem size each iteration to bring about the worst case if the sort4() does not have a randomized pivot. Table 4(a) shows that the time complexity in terms of big-Oh is O(N^2) and figure 4(a) shows this as well since selection sort would have the same pattern. To conclude sort4() is quicksort with no randomization. Diagram 1 is consistent with these findings.

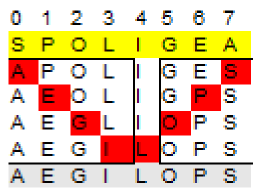
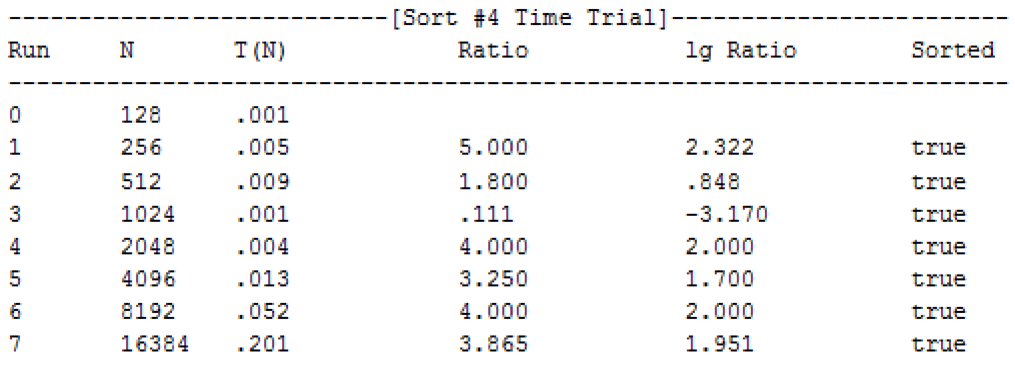
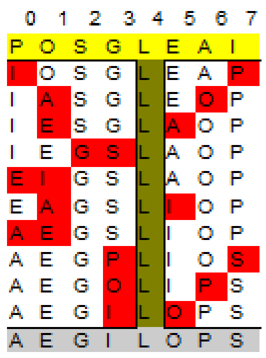


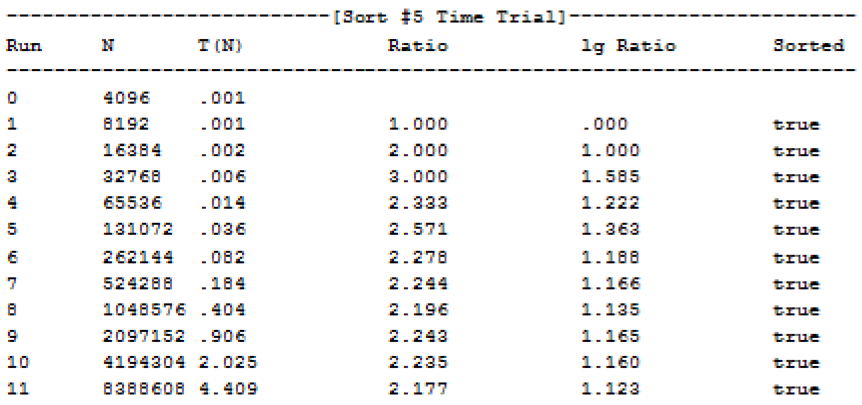
Figure 4(a) Worest Case visual representation.



```
---------------------------[Sort #4 Time Trial]------------------------
Run      N        T(N)             Ratio            lg Ratio          Sorted
-----------------------------------------------------------------------
0        128      .001
1        256      .005             5.000            2.322             true
2        512      .009             1.800            .848              true
3        1024     .001             .111             -3.170            true
4        2048     .004             4.000            2.000             true
5        4096     .013             3.250            1.700             true
6        8192     .052             4.000            2.000             true
7        16384    .201             3.865            1.951             true
```

Table 4(a). Worst Time trial for sort4()





```
---------------------------[Sort #5 Time Trial]------------------------
Run      N        T(N)             Ratio            lg Ratio          Sorted
-----------------------------------------------------------------------
0        4096     .001
1        8192     .001             1.000            .000              true
2        16384    .002             2.000            1.000             true
3        32768    .006             3.000            1.585             true
4        65536    .014             2.333            1.222             true
5        131072   .036             2.571            1.363             true
6        262144   .082             2.278            1.188             true
7        524288   .184             2.244            1.166             true
8        1048576  .404             2.196            1.135             true
9        2097152  .906             2.243            1.165             true
10       4194304  2.025            2.235            1.160             true
11       8388608  4.409            2.177            1.123             true
```

Figure 5.Visual representation of sort5() given an array of Strings {P,O,S,G,L.E.A.I}. Where the first row is identifies the element number, the second row, highlighted yellow, identifies the starting array, the last row identifies the array in natural order, highlighted gray, the red highlights indicate the modifications of the array.

Table 5. The first segment of data that was generated by the source code to find the big-Oh of method sort5(), there are five different columns: Run, N, T(N), Ratio, lg Ratio. Each column represents essential data needed to discover the time complexity which is the big-Oh.

To identify the sorting algorithm that is being applied by the method sort5() of SortTrials.class one can look at figure 5. The figure shows that the element does not move similar to figure 2, but unlike figure 2 the first modification of the array did not leave a natural order element in place. By leaving the element 'L' in place throughout the duration of the sort, we can think that this maybe a pivot point in the array. Give that 'L' is the pivot point and observation is made that the goal from the modifications is to place elements that are less then the pivot to the left and items greater than the pivot to the right and thus sort the array. By this we can conclude that the sorting algorithm being used is quicksort. To find out if it is quicksort with no randomization or an randomized quicksort, we have to do another test for worst case to see if sort5() is with randomization of the pivot. Figure 5(a) shows the results similar to the selection sort if given the same array. Table 5(a) revels that the quicksort algorithm used is with randomization of the pivot to prevent the worst case majority of the time. The time complexity of this result is O(N Log N). Diagram 1 confirms this result.

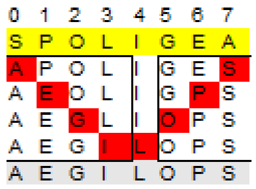
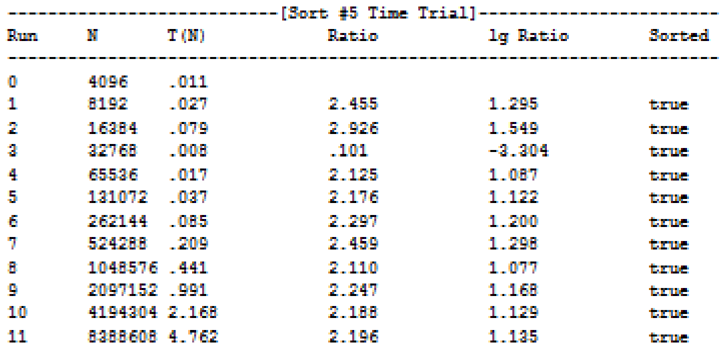


Figure 5(a) Worst case for sort5().



```
----------------------------[Sort #5 Time Trial]------------------------
Run      N         T(N)            Ratio            lg Ratio        Sorted
-------------------------------------------------------------------------

0        4096      .011
1        8192      .027            2.455            1.295           true
2        16384     .079            2.926            1.549           true
3        32768     .008            .101             -3.304          true
4        65536     .017            2.125            1.087           true
5        131072    .037            2.176            1.122           true
6        262144    .085            2.297            1.200           true
7        524288    .209            2.459            1.298           true
8        1048576 .441              2.110            1.077           true
9        2097152 .991              2.247            1.168           true
10       4194304 2.168             2.188            1.129           true
11       8388608 4.762             2.196            1.135           true
```

Table 5(a) Worst case for sort(5).

Conclusion

Figure 6 shows graphically the time vs problem size of each sort algorithm being used. The graph does not consider the worst case for quicksort. In conclusion with the results that were discussed earlier and to refresh each discussion, sort 1 is insertion sort, sort 2 is selection sort, sort 3 is merge sort, sort 4 is quick sort with out randomization, sort 5 is quick sort with randomization. Each sorting algorithm has its own unique pattern that distinguish it from the others.

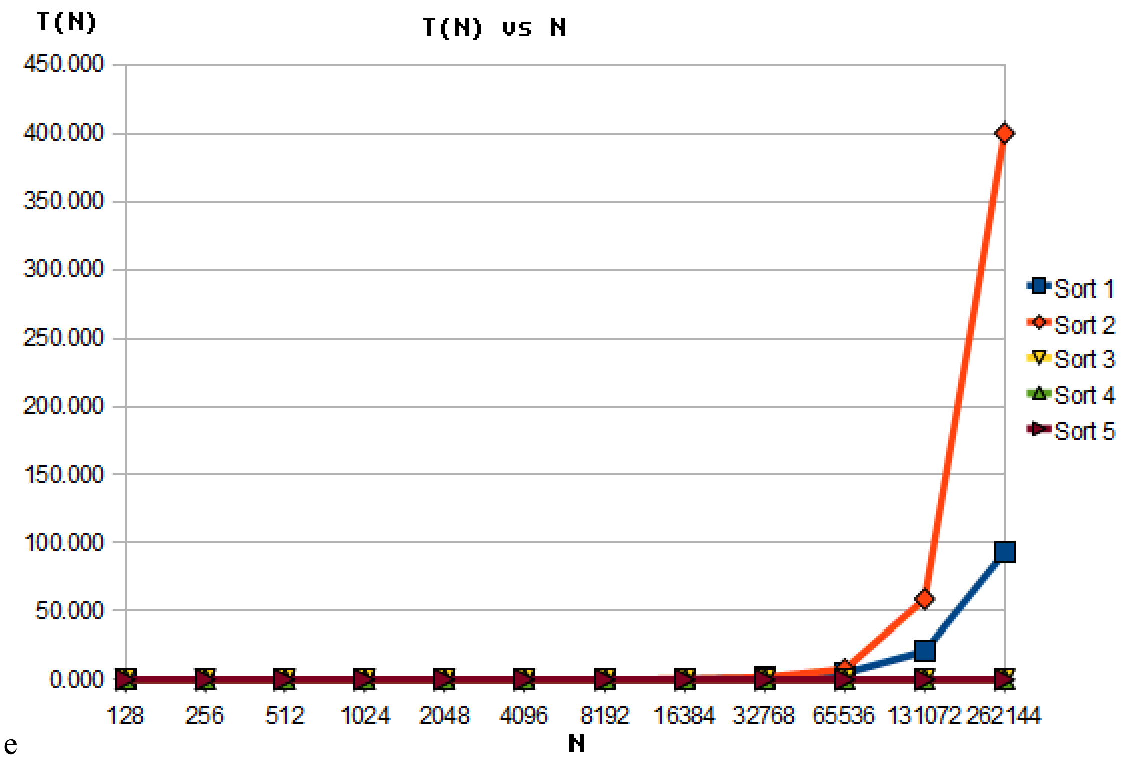


Figure 6. Shows the graph of each sort.

Appendices

The source code and the output of the program is located beyond this page.

```java
import java.text.DecimalFormat;

/**
 * SortTrialsClient.java. When executed runs a time trial
 * on each sorting method of SortTrials.class and prints
 * the result of the times to the screen in a tabular format.
 *
 *
 * Compilation:   %javac SortTrialsClient.java
 * Execution:     %java -ea SortTrialsClient
 *
 * @author    Walter Conway (wjc0008@auburn.edu)
 * @version   2013-02-10
 *
 */
 public final class SortTrialsClient {

    public static void main(String[] args) {
      Clock clock;              // measures elapsed time
      double elapsedTime = 0;   // elapsed time of current run
      double prevTime = 0;      // elapsed time of previous run
      double ratio = 0;         // currentTime / prevTime
      double lgratio = 0;       // log base 2 of ratio
      int N;                    // problem size parameter
      Integer[] b;              // problem size array
      DecimalFormat deci = new DecimalFormat("#.000");
      SortTrials<Integer> st = new SortTrials<Integer>();
      for (int t = 1; t < 6; t++) {

         switch (t) {
            case 3:
            case 4:
            case 5:
               N = 4096;
               b = populateNWC(N);
               break;
            default:
               N = 128;
               b = populateN(N);
         }

         System.out.println("---------------------------[Sort #" + t
                             + " Time Trial]------------------------");
         System.out.println("Run\tN\tT(N)\t\tRatio\t\tlg Ratio\tSorted");
         System.out.println("---------------------------------------"
                            + "--------------------------------");
         for (int i = 0; i < 10; i++) {
         //Comment out shuffle when running worst case for sort 4 & 5.
            st.shuffle(b);
            clock = new Clock();
            switch (t) {
               case 1:
                  st.sort1(b);
                  break;
               case 2:
                  st.sort2(b);
                  break;
               case 3:
                  st.sort3(b);
                  break;
               case 4:
                  st.sort4(b);
                  break;
               case 5:
                  st.sort5(b);
                  break;
               default:
            }
            elapsedTime = clock.elapsedTime();
            System.out.print(i + "\t" + N + "\t" + deci.format(elapsedTime));
            if (i != 0) {
               ratio = (elapsedTime / prevTime);
               lgratio = (Math.log(ratio) / Math.log(2));
               System.out.print("\t\t" + deci.format(ratio) + "\t\t"
                                   + deci.format(lgratio) +"\t\t"
                                   + st.isSorted(b) +"\n");
            }
            else {
               System.out.println();
            }

            b = populateN(N *= 2);
            prevTime = elapsedTime;

         }
         System.out.println();
      }
   }
   /**
   *  Used to populate N to have a different
   *  problem size each time
   */

    public static Integer[] populateN(int N) {
      Integer[] b = new Integer[N];
      for (int i = 0; i < N; i++) {
         b[i] = i;
      }
      return b;
   }
   /**
   *  Used to populate N to have a different
   *  problem size each time, but for worst
   *  case test.
   */
    public static Integer[] populateNWC(int N) {
      Integer[] b = new Integer[N];
      for (int i = 0; i < N; i++) {
         b[(N-1)-i] = i;
      }
      return b;
   }
```

```

    }
```

```
---------------------------[Sort #1 Time Trial]-----------------------
Run     N       T(N)            Ratio           lg Ratio        Sorted
----------------------------------------------------------------------
0       128     .002
1       256     .004            2.000           1.000           true
2       512     .012            3.000           1.585           true
3       1024    .006            .500            -1.000          true
4       2048    .003            .500            -1.000          true
5       4096    .019            6.333           2.663           true
6       8192    .074            3.895           1.962           true
7       16384   .295            3.986           1.995           true
8       32768   1.188           4.027           2.010           true
9       65536   4.823           4.060           2.021           true
10      131072  20.634          4.278           2.097           true
11      262144  93.065          4.510           2.173           true
---------------------------[Scrt #2 Time Trial|-----------------------
Run     N       T(N)            Ratio           lg Ratio        Sorted
----------------------------------------------------------------------
0       128     .0C1
1       256     .0C2            2.000           1.000           true
2       512     .0C6            3.000           1.585           true
3       1024    .0C2            .333            -1.585          true
4       2048    .0C7            3.500           1.807           true
5       4096    .023            3.286           1.716           true
6       8192    .090            3.913           1.968           true
7       16384   .360            4.000           2.000           true
8       32768   1.660           4.611           2.205           true
9       65536   7.606           4.582           2.196           true
10      131072  58.798          7.730           2.951           true
11      262144  40C.511         6.812           2.768           true
---------------------------[Sort #3 Time Trial]-----------------------
Run     N       T(N)            Ratio           lg Ratio        Sorted
----------------------------------------------------------------------

0       4096    .021
1       8192    .046            2.190           1.131           true
2       16384   .088            1.913           .936            true
3       32768   .011            .125            -3.000          true
4       65536   .023            2.091           1.064           true
5       131072  .043            1.870           .903            true
6       262144  .103            2.395           1.260           true
7       524288  .235            2.282           1.190           true
8       1048576 .561            2.387           1.255           true
9       2097152 1.328           2.367           1.243           true
10      4194304 2.787           2.099           1.069           true
11      8388608 6.400           2.296           1.199           true
---------------------------[Sort #4 Time Trial]-----------------------
Run     N       T(N)            Ratio           lg Ratio        Sorted
----------------------------------------------------------------------

0       4096    .006
1       8192    .006            1.000           .000            true
2       16384   .002            .333            -1.585          true
3       32768   .006            3.000           1.585           true
4       65536   .012            2.000           1.000           true
5       131072  .030            2.500           1.322           true
6       262144  .066            2.200           1.138           true
7       524288  .129            1.955           .967            true
8       1048576 .297            2.302           1.203           true
9       2097152 .686            2.310           1.208           true
10      4194304 1.444           2.105           1.074           true
11      8388608 3.327           2.304           1.204           true
---------------------------[Sort #5 Time Trial]-----------------------
Run     N       T(N)            Ratio           lg Ratio        Sorted
----------------------------------------------------------------------

0       4096    .001
1       8192    .001            1.000           .000            true
2       16384   .002            2.000           1.000           true
3       32768   .006            3.000           1.585           true
4       65536   .014            2.333           1.222           true
5       131072  .036            2.571           1.363           true
6       262144  .082            2.278           1.188           true
7       524288  .184            2.244           1.166           true
8       1048576 .404            2.196           1.135           true
9       2097152 .906            2.243           1.165           true
10      4194304 2.025           2.235           1.160           true
11      8388608 4.409           2.177           1.123           true
```