SORTING ALGORITHMS

Masters & Higher Diploma (Software Design and Development)
Catherine Gaughan-Smith 13232653

Contents

Catherine Gaughan-Smith
Student ID : 13232653

PART 1 High Level Pseudocode of Main Application showing inputs and how they impact the flow of control of the program.

MAIN PROGRAM : SortUtility class

```
**********************************************************
Mainline:
**********************************************************

BEGIN
        SET trace // to control the level of detail to print to console screen
        SET outputFileName to  "Output.txt"
        WRITE "Enter the name of the input file:-"
        READ filename

        SET option =  DO getSortOption

        IF option is not -1
                SET sortArray = DO readFromFile with parameter (filename)

                DO doSort with parameters( sortArray, option)

                DO writeToFile with parameters( outputFileName, sortArray)
        ENDIF
END


**********************************************************
Process:     getSortOption
**********************************************************
WRITE the option number for each sort algorithm
DO UNTIL option entered is valid
        WRITE "Please Enter option:"
        READ option
ENDDO

IF option is -1
        WRITE "Program Terminated"
ENDIF

RETURN option


**********************************************************
Process:     readFromFile
Input Parameter: filename
**********************************************************
Initialize arraylist
Initialize input stream for filename
IF file not found
        WRITE "File Not Found"
ELSE
        WHILE input stream not end of file
                READ integer from input stream
                SET next arraylist element to integer
        ENDWHILE
ENDIF
SET array = sequence of integers in arraylist
RETURN array
```

Catherine Gaughan-Smith
Student ID : 13232653

```
*********************************************************
Process:    doSort
Input Parameter: array, option
*********************************************************
IF         option = 1
        CALL doBubbleSort with parameter (sortArray)
ELSE  IF option = 2
        CALL doSelectionSort with parameter (sortArray)
ELSE  IF option = 3
        CALL doInsertionSort with parameter (sortArray)
ELSE  IF option = 4
        CALL doMergeSort with parameter (sortArray)
ELSE  IF option = 5
        CALL doQuickSort with parameter (sortArray)
ENDIF

SET isSorted = DO validate(sortArray)
IF isSorted is true
        WRITE “Sorted”
ELSE
        WRITE “Sort Failed”
ENDIF
RETURN

*********************************************************
Process:    writeToFile
Input Parameter: outputFileName, sortArray
*********************************************************
Initialize file to outputFileName
IF file not found
        WRITE “File Not Found”
ELSE
        WHILE array not end of file
                READ integer from sortArray
                SET next output stream element to integer
        ENDWHILE
        Set written flag to true
        Close output stream
ENDIF

IF written flag is true
        WRITE “Sorted data has been written to output file”
ELSE
        WRITE “Data was not written”
ENDIF
RETURN
```

PART 2 High Level Pseudocode of Main Application showing inputs and how they impact the flow of control of the program.

2.1 BubbleSort Algorithm

BubbleSort.java

```
8    @Override
9    public void sort(int[] sortMe) {
10
11       int endSeq = sortMe.length;
12
13       while (endSeq > 1) {
14           // after each pass the highest value of the pass relocated
15           // to the end of the pass, and the pass is decreased by 1
16           for (int idx = 0; idx < endSeq - 1; idx++) {
17               // compare each integer with it's neighbour and swap as needed
18               // to put the higher value to the right of the pair
19               if (sortMe[idx] > sortMe[idx + 1]) {
20                   swap(sortMe, idx, idx + 1);
21               }
22           }
23           // If trace on: Print details of sort as progresses thru each pass
24           if (trace) {
25               System.out.printf("      Pass %2d>  %s\n", pass++,
26                       arrayToString(sortMe));
27           }
28           endSeq--; // decrease the pass
29       }
30   }
```

In sorting algorithms, a "pass" is defined as one full trip through the array comparing and if necessary, swapping elements.

Line 13 Starts the loop counter at the length of the array to sort. The loop will decrease by 1 on each pass until the counter is one, since one element in an array is inherently sorted.
Line 16 Starting at the beginning of the array (with the element at index 0), compare each element with the element to the right. **Line 19** If the left of the pair of elements is greater then swap the pair, bringing the highest element forward to the right.
After each pass (from index 0 to the end : endsSeq-1) the highest value found is relocated at the end, and it is not reached again as the length of the pass decreases by one **Line 28**

2.2 SelectionSort Algorithm

```
6    /**
7     * Selection sort algorithm
8     */
9    public void sort(int[] sortMe) {
10
11       int nextSmallest;
12       boolean isSwap;
13
14       // each loop advance the start position by 1
15       for (int sortIdx = 0; sortIdx < sortMe.length - 1; sortIdx++) {
16           isSwap = false; // initialize swap required flag
17           nextSmallest = sortIdx; // initialize index of smallest value
18
19           // find the index of the next smallest value in the list
20           for (int i = sortIdx + 1; i < sortMe.length; i++) {
21
22               if (sortMe[i] < sortMe[nextSmallest]) {
23                   nextSmallest = i;
24                   isSwap = true;
25               } // end if
26           }// end for
27
28           if (isSwap) { // only swap if smaller value found
29               swap(sortMe, sortIdx, nextSmallest);
30           } // swap smallest found with element in position sortIdx.
31
32           // Print details of how selection sort progresses
33           if (trace) {
34               System.out.printf("%14s  %s\n", String.format("Select %2d>",sortMe[sortIdx]),
35               arrayToString(sortMe));
36           }
37       } // end for loop
38   } // end method sort
```

The **selection sort** is a combination of searching and sorting.
During each pass, the unsorted element with the smallest value is moved to its proper position in the array.

Line 15 Start the outer loop to traverse the array once. Each iteration of the outer loop is the starting index of the inner loop. **Line 20** The inner loop finds the next smallest value in the array. After checking each element to the right of the start position, if a value is found that is smaller then the value at the start position then it is swapped to bring the next smallest value to the start.
After each pass the next smallest value is relocated next to the previous smallest value, and it is not reached again as the start position is increased by one (by the outer loop).

2.3 InsertionSort Algorithm

```
5      /**
6       * Insertion sort algorithm
7       */
8      public void sort(int[] sortMe) {
9
10         int sortValue; // integer to be sorted
11         int insertIdx; // index of integer to be sorted
12
13         // Select the next integer to be inserted
14         // relative the all items to the left of integer
15         for (int i = 1; i < sortMe.length; i++) {
16
17             sortValue = sortMe[i];
18             insertIdx = i;
19             // find the correct insertion point
20             for (int j = i; j > 0; j--) {
21                 if (sortMe[j - 1] > sortValue) {
22                     sortMe[j] = sortMe[j - 1];
23                 } else {
24                     break;
25                 }
26                 insertIdx--;
27             }
28             sortMe[insertIdx] = sortValue;
29             // swap out
30             // Print details of how bubble sort progresses
31             if (trace) {
32                 System.out.printf("%14s  %s\n", String.format("Insert %3d>",sortValue),
33                 arrayToString(sortMe));
34             } //end if
35         } // end for loop
36     } // end sort method
37 } // end InsertionSort class
```

The **insertion sort** passes through the array only once.
As it traverses the array all elements to the left of the loop counter are in sorted order.

Line 15 Sets up the outer loop to traverse the array once. **Line 17** Sets `sortValue` to of the next value that needs to be inserted in the correct position.
Line 20 Starts the inner loop which reads back down the sorted set of elements to the left to find the correct insertion point for the `sortValue.` **Line 22** As it steps back each element is shuffled one place to the right to allow the `sortValue` to be inserted.
Line 24 The inner loop terminates when the element to the left is less then or equal to the `sortValue`, which indicates that the insertion point is found: `insertIdx.`
Line 28 The `sortValue` is inserted at its required position, at `insertIdx`.
The loop continues to the next element in the array, until it finishes sorting the last element.

2.4 QuickSort Algorithm

```
6       /**
7        * Quick sort algorithm
8        */
9       public void sort(int[] sortMe) {
10
11          int select1 = 0;
12          int select2 = sortMe.length - 1;
13          partition(sortMe, select1, select2);
14
15          if (trace) {
16              System.out.println();
17          }
18
19      } // end sort method
20
21      // Quick sort Partition Method : Recursive
22      private void partition(int[] splitArray, int low, int high) {
23          int pivot = splitArray[(int) (Math.floor(Math.random()
24                  * (high - low + 1)) + low)];
25
26          int i = low;
27          int j = high;
28
29          if (trace) { // Print trace details following each partition
30              System.out.printf("\nBetween(%2d,%2d) pivot %4d> %s", low, high,
31                      pivot, arrayToString(splitArray));
32          }
33
34          while (i <= j) {
35              while (splitArray[i] < pivot) i++; //from left stop  at >= pivot
36              while (splitArray[j] > pivot) j--; //from right stop at <= pivot
37
38              // Swap two elements
39              if (i < j) {
40                  swap(splitArray, i, j);
41
42                  // Print trace details of how quick sort progresses
43                  if (trace) {
44                      System.out.printf(", swap %d and %d", splitArray[i],
45                              splitArray[j]);
46                  } // end trace
47              } // end swap condition
48
49              if (i <= j) {
50                  i++;
51                  j--;
52              } // iterate converging indexes
53          }
54
55          if ((high - low) <= 1) return; // arrays of 2 elements are sorted
56
57          if (low < j)    partition(splitArray, low, j);
58          if (i < high)   partition(splitArray, i, high);
59
60      } // end partition method
61  }
62
```

The **quick sort** is a divide and conquer algorithm.
It splits the array at a random element, called the pivot, and then swaps all other elements right of the pivot if they are greater then the pivot value, or else to the left of the pivot.
Then the two subsets of elements delimited by the pivot go through the same process, until the size of the subset to be sorted is less then 2 (since one element in a subset is deemed 'sorted')

Line 13 Start the sort with a call to a recursive method `partition` which takes three parameters : the array to be sorted, the start index `low` and the end index `high` which delimit the portion of the array to be sorted. The first time this method is called the portion to be sorted is the entire array. Subsequent calls to the `partition` method will process increasingly smaller portions of the array.
Line 35 and 36 The `low` index in incremented until a value greater than or equal to the pivot value is found. The `high` index in decremented until a value less than or equal to the pivot value is found. Whilst the low and high indexes have not converged, the value to the left is swapped with the value to the right, in order to place elements right of the pivot if they are greater then the pivot value, or else to the left of the pivot.
Line 55 returns from the recursive calls to the `partition` method if the size of the portion of the array is less then two. Otherwise the `partition` method is called again with two subsets of the current portion, i.e. those element greater then the pivot value and then the second subset, those elements less than or equal to the pivot value.

2.5 MergeSort Algorithm

```
1  package SortAlgorithms;
2
3  public class MergeSort extends Sort{
4
5      /**
6       * Merge sort algorithm
7       */
8      public void sort(int[] sortMe) {
9
10         // copy values into sortMe from new sorted array
11         // so that changes propagate back to calling method
12         int[] sorted = merge(sortMe);
13         for (int i = 0; i < sortMe.length; i++) {
14             sortMe[i] = sorted[i]; // affect change to original array
15         }
16
17     } // end MergeSort method
18
19     // Mergesort Merge Method : Recursive
20     private int[] merge(int[] mergeArray) {
21         // if length of array is 1 then considered sorted
22         if (mergeArray.length <= 1)
23             return mergeArray;
24
25         // split into smaller arrays on each recursive call
26         int[][] parts = splitArray(mergeArray);
27
28         // when all parts return length 1 then recombine with
29         // process mergeArrays
30         return mergeArrays(merge(parts[0]), merge(parts[1]));
31     }
```

The **Merge sort** is also a divide and conquer algorithm.
It splits the array into subsets of one or two elements.
Then it recombines each subset with one other to create a sorted subset. It finishes with combining the last two subsets into one whole sorted array.

Line 12 Start the sort with a call to a recursive method `merge` with successively smaller subset of the original array as a parameter, and return a sorted set.
Line 13 Populates the original array with the values in the sorted set.
Line 26 Calls the method `splitArray` which takes the current portion of the array, splits it in half and returns two subsets.
Line 30 Calls the method `mergeArray` which reconstitutes the two subsets into a single sorted array.

```
33      // Mergesort Split Array in half to get two arrays
34      private int[][] splitArray(int[] splitMe) {
35          int[][] splitter = new int[2][];
36          int splitHalf = (int) splitMe.length / 2;
37
38          splitter[0] = new int[splitHalf];
39          splitter[1] = new int[splitHalf + splitMe.length % 2];
40
41          // split arrays
42          for (int i = 0; i < splitHalf; i++) {
43              splitter[0][i] = splitMe[i];
44              splitter[1][i] = splitMe[i + splitHalf];
45          }
46          if (splitMe.length % 2 > 0)
47              splitter[1][splitHalf] = splitMe[splitHalf * 2];
48
49          return splitter;
50      }
51
52      // Merge two sorted arrays into one sorted array
53      private int[] mergeArrays(int[] arryA, int[] arryB) {
54
55          int[] combiner = new int[arryA.length + arryB.length];
56
57          int j = 0;
58          int k = 0;
59          for (int i = 0; i < combiner.length; i++) {
60              if (!(k < arryB.length)
61                      || (j < arryA.length && arryA[j] <= arryB[k]))
62                  combiner[i] = arryA[j++];
63
64              else if (!(j < arryA.length)
65                      || (k < arryB.length && arryB[k] < arryA[j]))
66                  combiner[i] = arryB[k++];
67          }
68          // Print trace details of how merge sort progresses
69          if (trace) {
70              System.out.printf("%2d> Merge: %s and %s resulting in: %s\n",
71                      pass++, arrayToString(arryA), arrayToString(arryB),
72                      arrayToString(combiner));
73          }
74
75          return combiner;
76      } // end method mergeArrays
77  }
78
```

method splitArray

Line 35 The variable `splitter` is a two dimensional array the is returned from the `splitArray` method.

method mergeArray

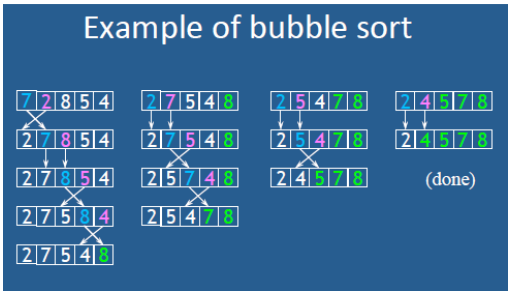
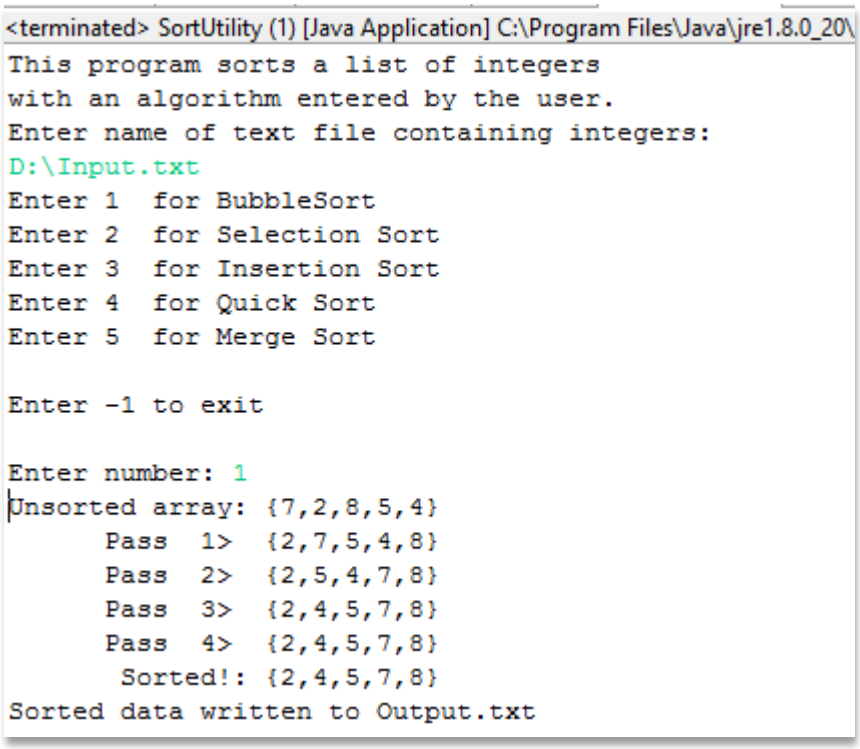
Line 55 The variable `combiner` is an array created to take the sorted elements of the two arrays passed to the method `mergeArray`.
Line 59 Loop through the combiner array and for each index, populate the `combiner` element whin the next highest value either from the first or second array arguments.
Line 76 Return the sorted array recursively until the whole array has been merged back together.

Catherine Gaughan-Smith
Student ID : 13232653

PART 3 Testing each sort implementation.

SET the class variable SortUtility.trace to true

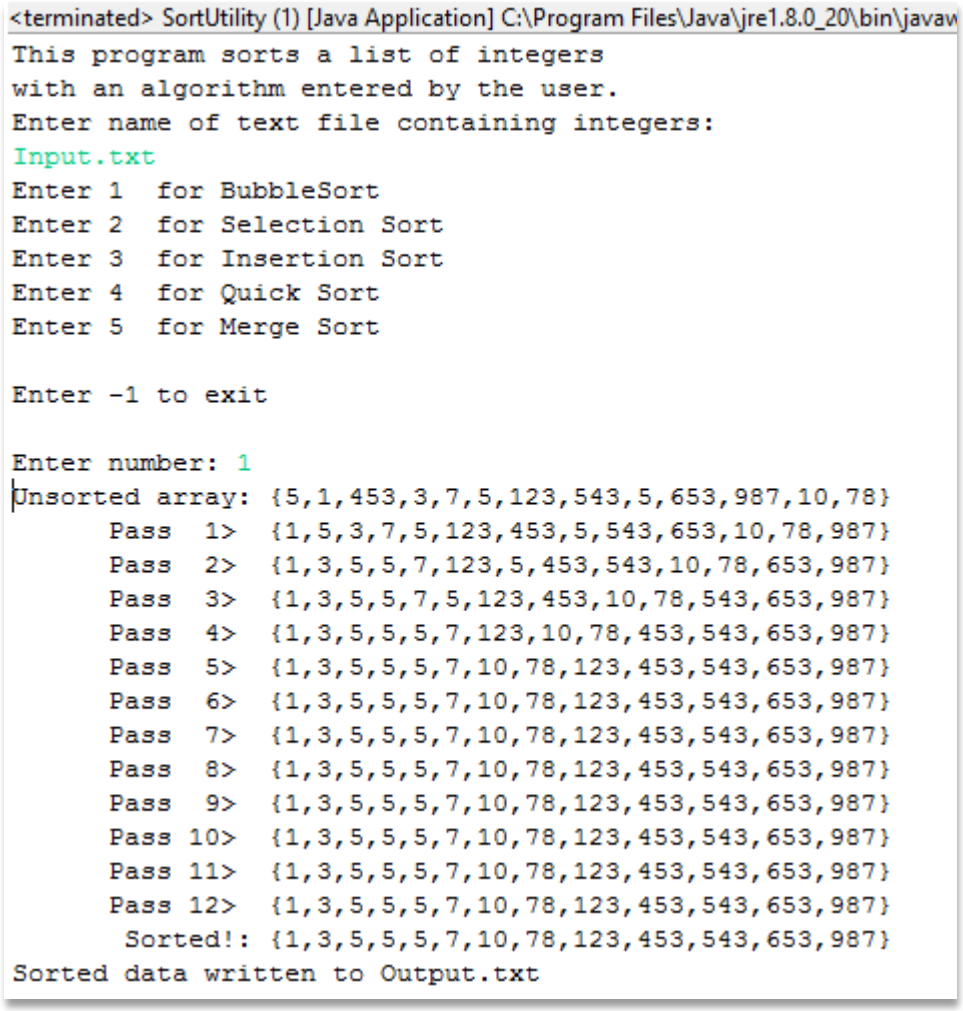
This flag controls the level of detail printed to the console when any of the sort algorithms is run.
When the flag is set to true, each step of the running sort is reported – showing how the sequence of integers progresses to a sorted set.
When the flag is set to false, the selected sort algorithm is run and no output is written to the console until the sort is completed and the message "Sorted" is written.

Test 1.0	**Bubble sort algorithm** **Pre-condition :** `SortUtility.trace` is set to true
Expected Result:	The output should follow the same sort progression as on the slide from the lecture
	 
Actual Result:	RESULT : each pass on the sort progression matches the corresponding column example on the lecture slide for BubbleSort Sorted : `{2,4,5,7,8}`

```
<terminated> SortUtility (1) [Java Application] C:\Program Files\Java\jre1.8.0_20\
This program sorts a list of integers
with an algorithm entered by the user.
Enter name of text file containing integers:
D:\Input.txt
Enter 1  for BubbleSort
Enter 2  for Selection Sort
Enter 3  for Insertion Sort
Enter 4  for Quick Sort
Enter 5  for Merge Sort

Enter -1 to exit

Enter number: 1
Unsorted array: {7,2,8,5,4}
      Pass  1>  {2,7,5,4,8}
      Pass  2>  {2,5,4,7,8}
      Pass  3>  {2,4,5,7,8}
      Pass  4>  {2,4,5,7,8}
       Sorted!: {2,4,5,7,8}
Sorted data written to Output.txt
```

Test 1.1	**Test BubbleSort with some duplicates numbers** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	This array will be sorted : `{5,1,453,3,7,5,123,543,5,653,987,10,78}`
BubbleSort	
Actual Result:	`Sorted result : {1,3,5,5,5,7,10,78,123,453,543,653,987}`

Test 1.2	**Bubble sort negative values**
Expected Result	This array will be sorted : `{10,-10,10}`
Actual Result:	Array sorted normally: `{-10,10,10}`

Test 1.3	**Bubble sort on a value bigger then int value**
Expected Result	This array will be sorted : `{2147483647, 2147483648}`
Actual Result:	The second value is too high for an int – it is ignored. Result : `{2147483647}`

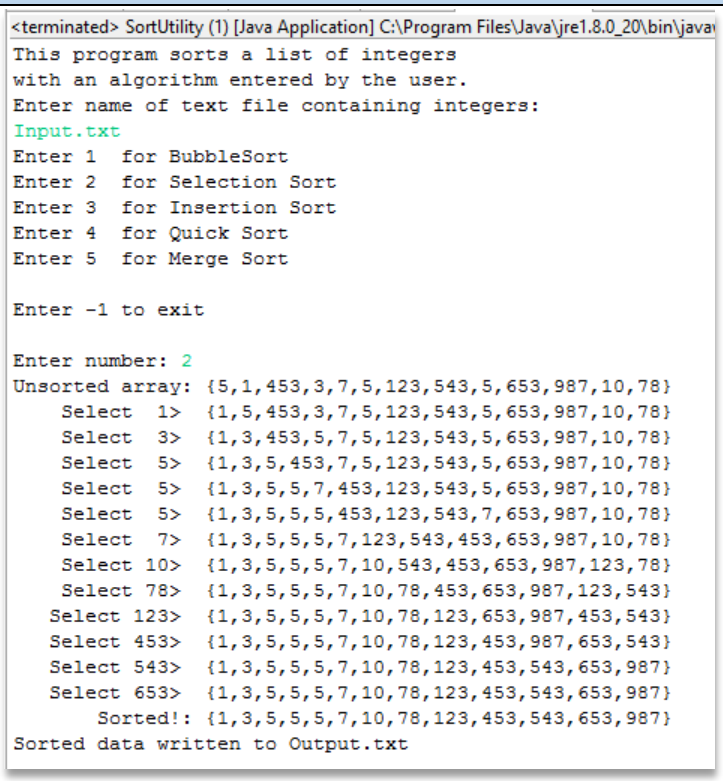
Test 1.4	**Bubble sort on all duplicate numbers** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	This array will be sorted : `{7,7,7,7,7}`

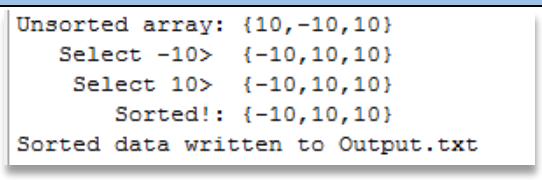
	`Unsorted array: {7,7,7,7,7}`
	`Pass 1> {7,7,7,7,7}`
	`Pass 2> {7,7,7,7,7}`
	`Pass 3> {7,7,7,7,7}`
	`Pass 4> {7,7,7,7,7}`
	`Sorted!: {7,7,7,7,7}`
	`Sorted data written to Output.txt`

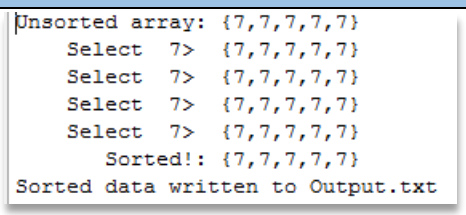
Actual Result:	Array sorted normally: `{7,7,7,7,7}`

Test 2.0	**Selection sort algorithm** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	The following sequence is sorted `{7,2,8,5,4}`

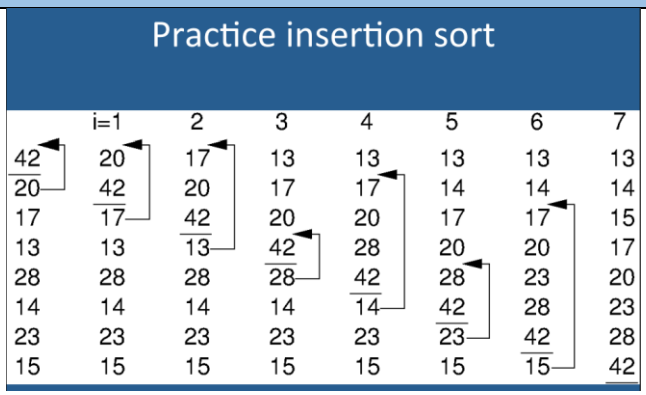
SelectionSort	
	• The selection sort might swap an array element with itself--this is harmless, and not worth checking for
	• Analysis:
	– The outer loop executes n-1 times
	– The inner loop executes about n/2 times on average (from n to 2 times)
	– Work done in the inner loop is constant (swap two array elements)
	– Time required is roughly (n-1)*(n/2)
	– You should recognize this as $O(n^2)$
	
	`<terminated> SortUtility (1) [Java Application] C:\Program Files\Java\jre1.8.0_20\b`
	`This program sorts a list of integers`
	`with an algorithm entered by the user.`
	`Enter name of text file containing integers:`
	`D:\Input.txt`
	`Enter 1 for BubbleSort`
	`Enter 2 for Selection Sort`
	`Enter 3 for Insertion Sort`
	`Enter 4 for Quick Sort`
	`Enter 5 for Merge Sort`
	
	`Enter -1 to exit`
	
	`Enter number: 2`
	`Unsorted array: {7,2,8,5,4}`
	`Select 2> {2,7,8,5,4}`
	`Select 4> {2,4,8,5,7}`
	`Select 5> {2,4,5,8,7}`
	`Select 7> {2,4,5,7,8}`
	`Sorted!: {2,4,5,7,8}`
	`Sorted data written to Output.txt`

Test 2.1	**Test Selection algorithm with some duplicates numbers** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	This array will be sorted : `{5,1,453,3,7,5,123,543,5,653,987,10,78}`
	
Actual Result:	`Sorted result : {1,3,5,5,5,7,10,78,123,453,543,653,987}`

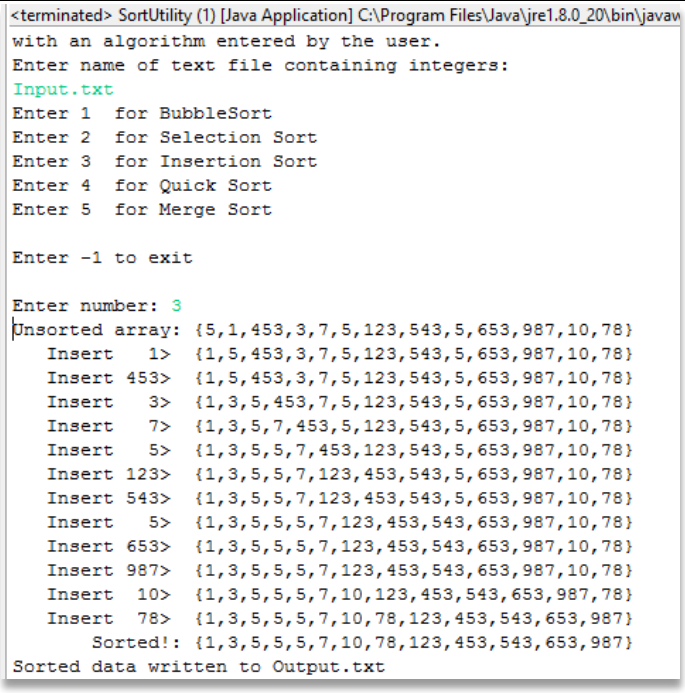
Test 2.2	**Selection sort negative values** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	This array will be sorted : `{10,-10,10}`
	
Actual Result:	Array sorted normally: `{-10,10,10}`

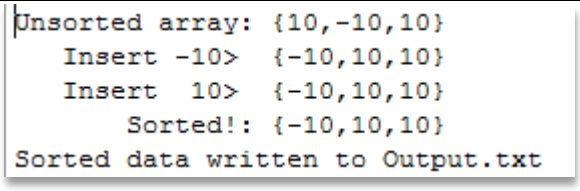
Test 2.3	**Selection sort on all duplicate numbers** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	This array will be sorted : `{7,7,7,7,7}`
	
Actual Result:	Array sorted normally: `{7,7,7,7,7}`

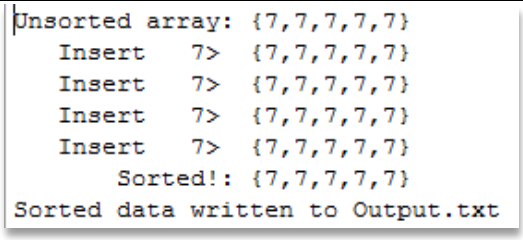
Test 3.0	**Insertion sort algorithm** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	The following sequence is sorted `{42,20,17,13,28,14,23,15}`

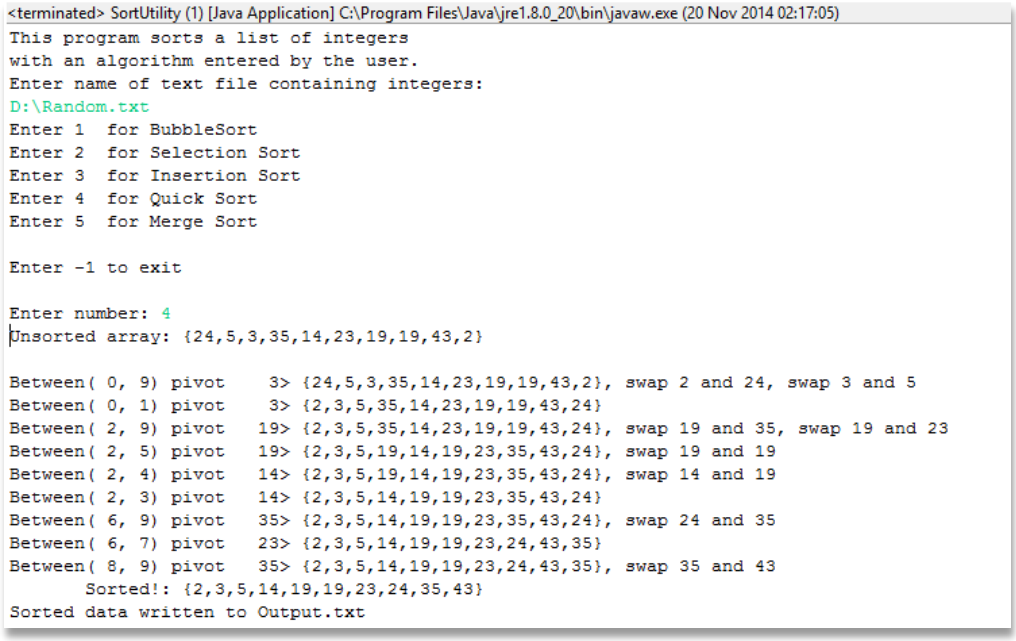
InsertionSort	
	
	`<terminated> SortUtility (1) [Java Application] C:\Program Files\Java\jre1.`
	`This program sorts a list of integers`
	`with an algorithm entered by the user.`
	`Enter name of text file containing integers:`
	`D:\Random.txt`
	`Enter 1  for BubbleSort`
	`Enter 2  for Selection Sort`
	`Enter 3  for Insertion Sort`
	`Enter 4  for Quick Sort`
	`Enter 5  for Merge Sort`
	
	`Enter -1 to exit`
	
	`Enter number: 3`
	`Unsorted array: {42,20,17,13,28,14,23,15}`
	`   Insert  20>  {20,42,17,13,28,14,23,15}`
	`   Insert  17>  {17,20,42,13,28,14,23,15}`
	`   Insert  13>  {13,17,20,42,28,14,23,15}`
	`   Insert  28>  {13,17,20,28,42,14,23,15}`
	`   Insert  14>  {13,14,17,20,28,42,23,15}`
	`   Insert  23>  {13,14,17,20,23,28,42,15}`
	`   Insert  15>  {13,14,15,17,20,23,28,42}`
	`       Sorted!: {13,14,15,17,20,23,28,42}`
	`Sorted data written to Output.txt`

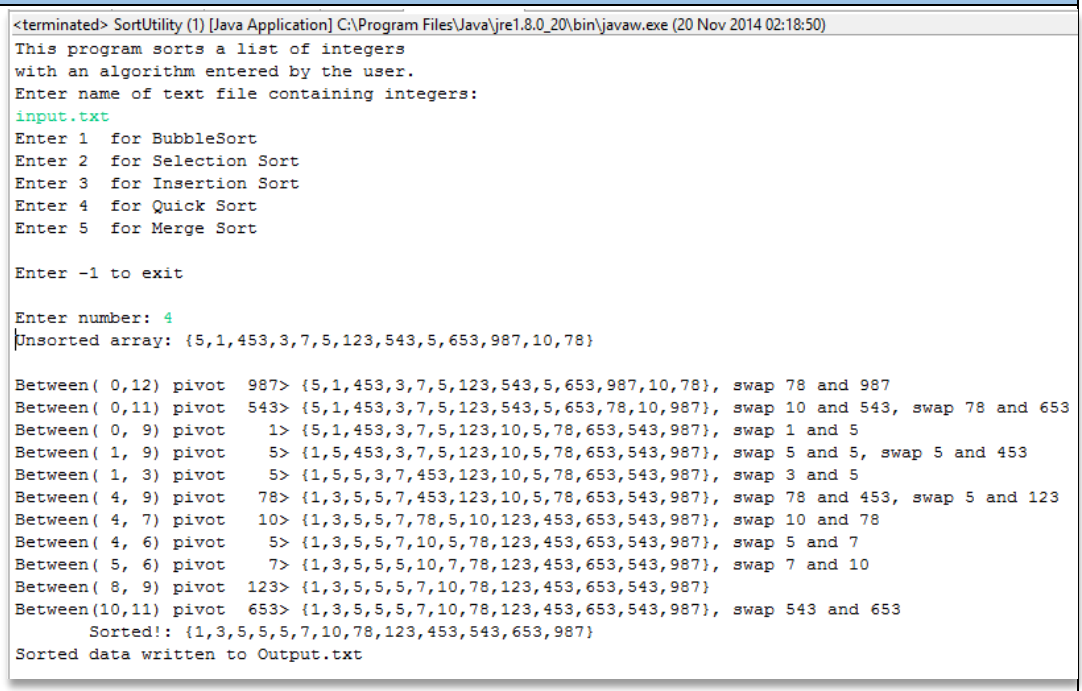
Actual Result:	Array sorted normally: `{13,14,15,17,20,23,28,42}`

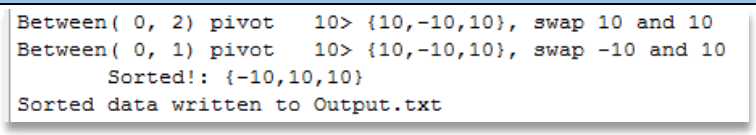
Test 3.1	**Test Insertion algorithm with some duplicates numbers** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	This array will be sorted : `{5,1,453,3,7,5,123,543,5,653,987,10,78}`
	
Actual Result:	`Sorted result : {1,3,5,5,5,7,10,78,123,453,543,653,987}`

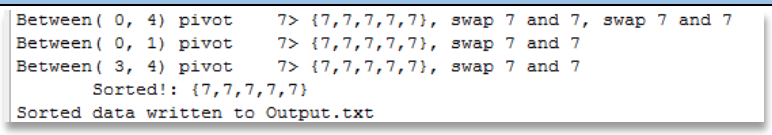
Test 3.2	**Insertion sort negative values** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	This array will be sorted : `{10,-10,10}`
	
Actual Result:	Array sorted normally: `{-10,10,10}`

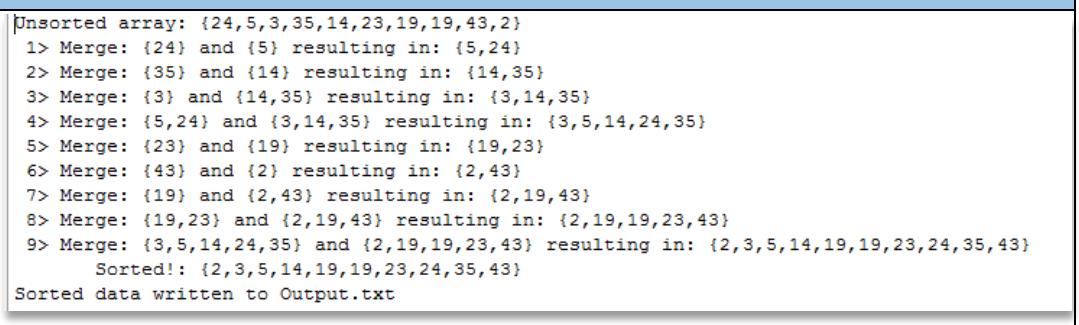
Test 3.3	**Insertion sort on all duplicate numbers** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	This array will be sorted : `{7,7,7,7,7}`
	
Actual Result:	Array sorted normally: `{7,7,7,7,7}`

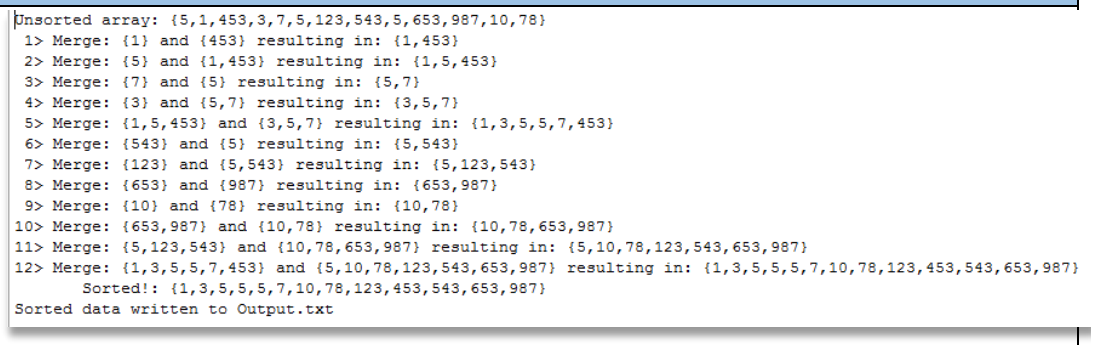
Test 4.0	**Quick sort algorithm** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	The following sequence is sorted `{24,5,3,35,14,23,19,19,43,2}`
QuickSort	
Actual Result:	Array sorted normally: `{2,3,5,14,19,19,23,24,35,43}`

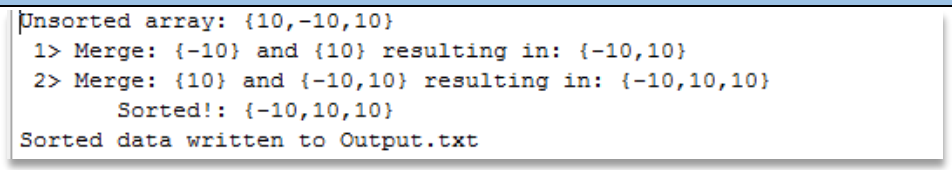
Test 4.1	**Test Quick algorithm with some duplicates numbers** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	This array will be sorted : `{5,1,453,3,7,5,123,543,5,653,987,10,78}`
	
Actual Result:	`Sorted result : {1,3,5,5,5,7,10,78,123,453,543,653,987}`

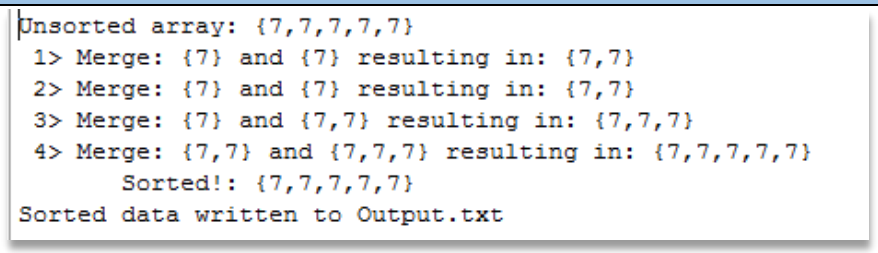
Test 4.2	**Quick sort negative values** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	This array will be sorted : `{10,-10,10}`
	
Actual Result:	Array sorted normally: `{-10,10,10}`

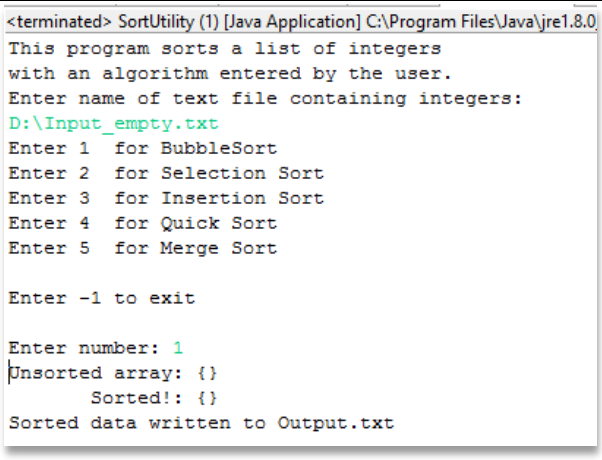
Test 4.3	**Quick sort on all duplicate numbers** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	This array will be sorted : `{7,7,7,7,7}`
	
Actual Result:	Array sorted normally: `{7,7,7,7,7}`

Test 5.0	**Merge sort algorithm** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	The following sequence is sorted `{24,5,3,35,14,23,19,19,43,2}`
MergeSort	
Actual Result:	Array sorted normally: `{2,3,5,14,19,19,23,24,35,43}`

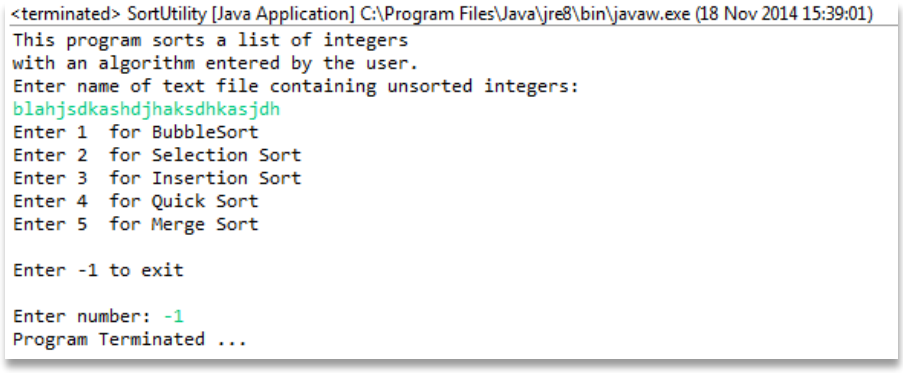
Test 5.1	**Test Merge algorithm with some duplicates numbers** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	This array will be sorted : `{5,1,453,3,7,5,123,543,5,653,987,10,78}`
	
Actual Result:	`Sorted result : {1,3,5,5,5,7,10,78,123,453,543,653,987}`

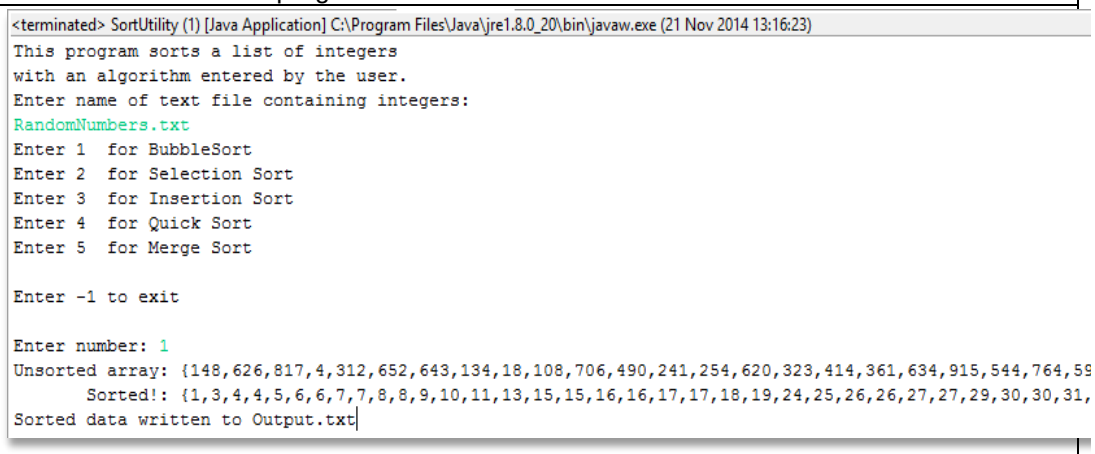
Test 5.2	**Merge sort negative values** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	This array will be sorted : `{10,-10,10}`
	
Actual Result:	Array sorted normally: `{-10,10,10}`

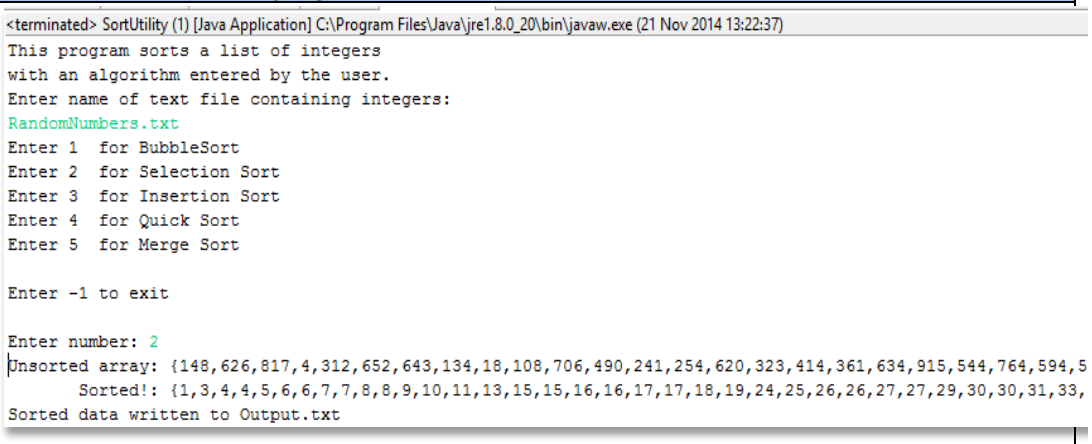
Test 5.3	**Merge sort on all duplicate numbers** **Pre-condition : `SortUtility.trace`** is set to true
Expected Result	This array will be sorted : `{7,7,7,7,7}`
	
Actual Result:	Array sorted normally: `{7,7,7,7,7}`

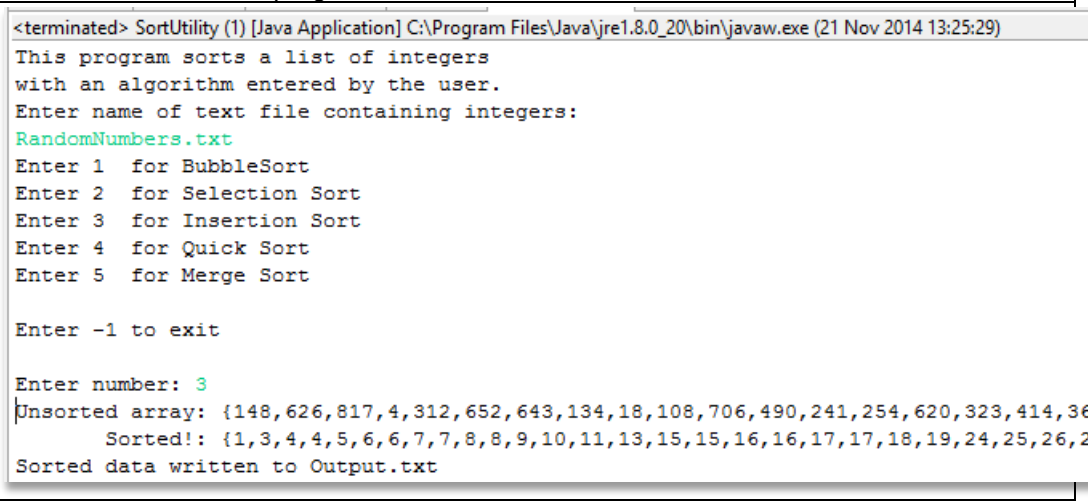
Test 6.0	**Input file is empty**
Expected Result	Program terminates without error
	

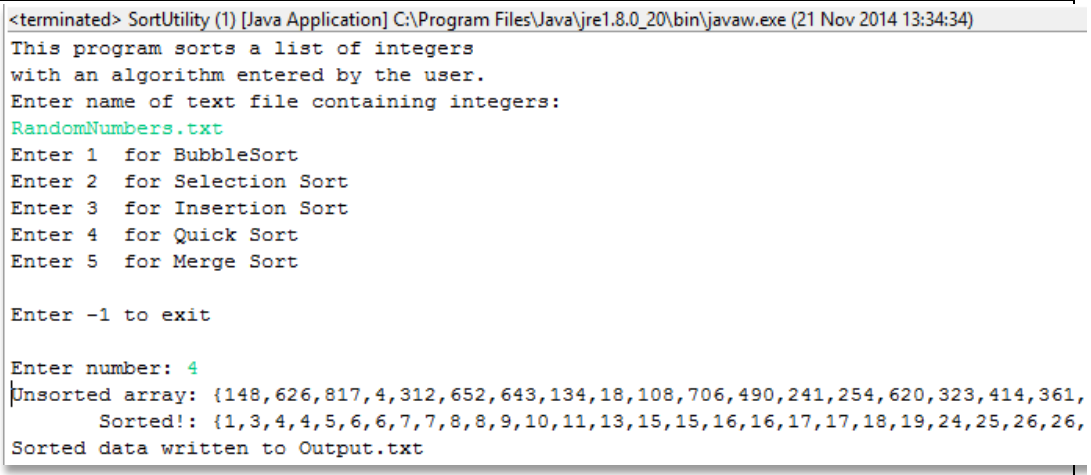
Test 6.1	**User terminates program**
Expected Result	When user enters -1 for option, the filename is not validated and the program terminates
Actual **Result**	Program ends normally

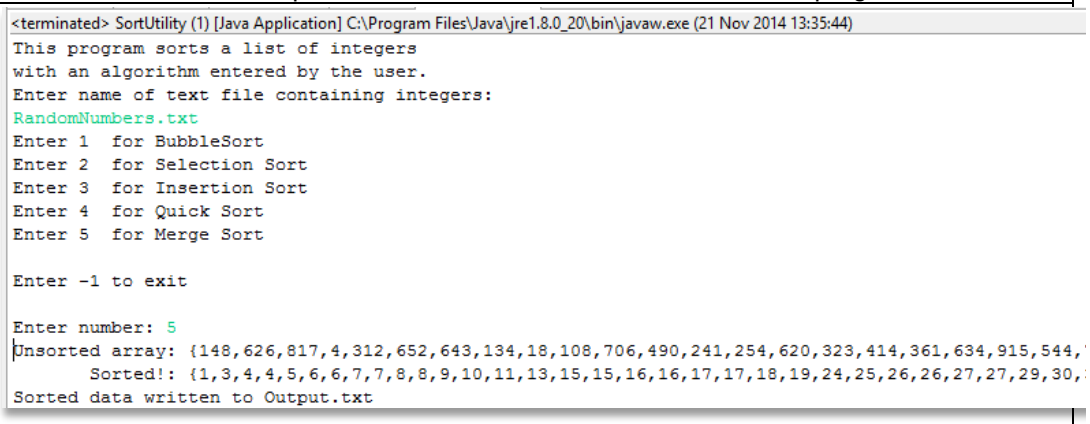


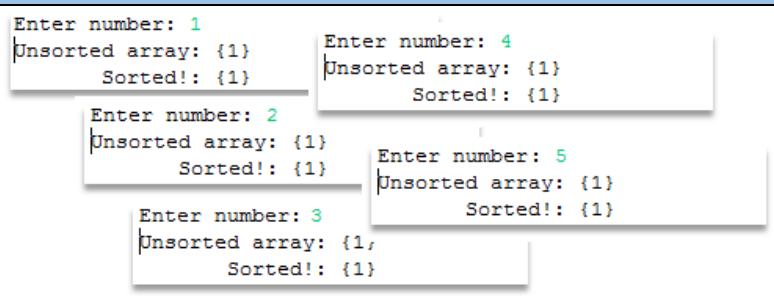
Test 7.0	**Sort** `RandomNumber.txt` **with BubbleSort** **Pre-condition :** `SortUtility.trace` is set to false
Expected Result	When user enters -1 for option, the filename is not validated and the program terminates
Actual **Result**	
Actual Result:	**BubbleSort** sorted `RandomNumber.txt`

Test 7.1	**Sort** `RandomNumber.txt` **with SelectionSort** **Pre-condition :** `SortUtility.trace` is set to false
Expected Result	When user enters -1 for option, the filename is not validated and the program terminates
Actual **Result**	
Actual Result:	**SelectionSort** sorted `RandomNumber.txt`

Test 7.2	**Sort** `RandomNumber.txt` **with InsertionSort** **Pre-condition :** `SortUtility.trace` is set to false
Expected Result	When user enters -1 for option, the filename is not validated and the program terminates
Actual **Result**	
Actual Result:	**InsertionSort** sorted `RandomNumber.txt`

Test 7.3	**Sort** `RandomNumber.txt` **with QuickSort** **Pre-condition :** `SortUtility.trace` is set to false
Expected Result	When user enters -1 for option, the filename is not validated and the program terminates
Actual **Result**	
Actual Result:	**QuickSort** sorted `RandomNumber.txt`

Test 7.4	**Sort** `RandomNumber.txt` **with MergeSort** **Pre-condition :** `SortUtility.trace` is set to false
Expected Result	When user enters -1 for option, the filename is not validated and the program terminates
Actual **Result**	
Actual Result:	**MergeSort** sorted `RandomNumber.txt`

Test 8.0	**Sort one element**
Expected Result	All sort algorithms can sort one element and end normally
	
Actual Result:	All sort algorithms sorted one element