Searching & Sorting in Java – Selection Sort

Using an insertion sort, values are constantly being moved. A sorting technique that reduces the amount of data movement is the *selection sort*. Once an item has been placed in its position in the ordered sequence, it is never moved again.

We can begin by scanning all of the values and finding the largest one. This value is placed at the top (end) of the list, exchanging positions with the item that was originally in that position.

On each subsequent pass, we examine progressively shorter lists, as the end values are already in their correct location.

1. Locate the largest value	6	3	5	8	2
2. Swap the largest value with the end of the list	6	3	5	2	8
3. Locate the largest remaining value	6	3	5	2	8
4. Swap this value with the end of the shortened list	2	3	5	6	8
5. Locate the largest remaining value	2	3	5	6	8
6. Swap with the end of the shortened list	2	3	5	6	8
7. Locate the largest remaining value	2	3	5	6	8
8. Swap with the end of the shortened list	2	3	5	6	8
9. With only one element left, sort is complete	2	3	5	6	8

To code this algorithm in Java, we note that, for an array called `list`, we must successively find the largest item in sublists of sizes `list.length`, `list.length-1`, etc..

```
for (int top = list.length – 1; top > 0; top--)
      // locate largest item and then
      // swap it with item at list[top]
```

The full method is as follows.

```
public static void selectSort (double[] list)
{
      for (int top = list.length – 1; top > 0; top--)
      {
            int largeLoc = 0;     // location of largest element
                                  // assume list[0] is largest to start
            for (int i = 1; i <= top; i++)   // check list[1] to list[top]
                  if (list[i] > list[largeLoc])
                        largeLoc = i;

            double temp = list[top];   // temporary storage
            list[top] = list[largeLoc];
            list[largeLoc] = temp;
      }
}
```

Exercises

1. If a selection sort were to be used to sort the data shown below in alphabetical order, show the data after each pass of the sort.

   Robert Brian Victor David Scott

2. In the `selectSort` method, what would happen if the expression `list[i] > list[largeLoc]` were to be changed to `list[i] < list[largeLoc]`?

3. In our version of selection sort, if the largest item is already at location `top` in the list, then the method still swaps that value with itself, even though it is not necessary.

   (a) How could the method be changed to avoid this unnecessary swapping?

   (b) Why might is be better to leave the method as it is?

4. On each pass of our version of selection sort, the *largest* value among the remaining unsorted items was placed in its correct position. An alternate form of the algorithm uses each pass to place the *smallest* value among the remaining unsorted values in its correct position.

   (a) Given the set of data

   8 9 6 1 2 4

   show the data as they would appear after each pass of a selection sort that moves the *smallest* data.

   (b) Write a Java method that implements this algorithm to sort an array of `int` values.

5. Sometimes we are only interested in knowing the values that would occupy one end of the list if the list were sorted. For example, we may want to know the scores of only the top ten competitors in a contest. Overload our selection sort method so that it puts the *k* largest values in order in the last *k* positions of the array. The value of *k* should be a parameter of the method.

```
public static void selectSort (double[] list, int k)
```