19.7 Insertion Sort

Insertion sort is another *simple, but inefficient*, sorting algorithm. The first iteration of this algorithm takes the *second element* in the array and, if it's *less than* the *first element*, *swaps it with the first element*. The second iteration looks at the third element and inserts it into the correct position with respect to the first two, so all three elements are in order. At the *i*th iteration of this algorithm, the first *i* elements in the original array will be sorted.

Consider as an example the following array, which is identical to the one used in the discussion of selection sort.

```
 34  56  4  10  77  51  93  30  5  52
```

A program that implements the insertion sort algorithm will first look at the first two elements of the array, 34 and 56. These are already in order, so the program continues. (If they were out of order, the program would swap them.)

In the next iteration, the program looks at the third value, 4. This value is less than 56, so the program stores 4 in a temporary variable and moves 56 one element to the right. The program then checks and determines that 4 is less than 34, so it moves 34 one element to the right. The program has now reached the beginning of the array, so it places 4 in the zeroth element. The array now is

```
   4   34   56   10   77   51   93   30   5   52
```

In the next iteration, the program stores 10 in a temporary variable. Then it compares 10 to 56 and moves 56 one element to the right because it’s larger than 10. The program then compares 10 to 34, moving 34 right one element. When the program compares 10 to 4, it observes that 10 is larger than 4 and places 10 in element 1. The array now is

```
   4   10   34   56   77   51   93   30   5   52
```

Using this algorithm, at the *i*th iteration, the first *i* elements of the original array are sorted, but they may not be in their final locations—smaller values may be located later in the array.

19.7.1 Insertion Sort Implementation

Class `InsertionSortTest` (Fig. 19.5) contains:

- `static` method `insertionSort` to sort `int`s using the insertion sort algorithm,
- `static` method `printPass` to display the array contents after each pass, and
- `main` to test method `insertionSort`.

Method `main` (lines 51–60) is identical to `main` in Fig. 19.4 except that line 58 calls method `insertionSort` to sort the array’s elements into ascending order.

```
 1  // Fig. 19.5: InsertionSortTest.java
 2  // Sorting an array with insertion sort.
 3  import java.security.SecureRandom;
 4  import java.util.Arrays;
 5
 6  public class InsertionSortTest {
 7     // sort array using insertion sort
 8     public static void insertionSort(int[] data)
 9        // loop over data.length - 1 elements
10        for (int next = 1; next < data.length; nex
11           int insert = data[next]; // value to in
12           int moveItem = next; // location to pla
13
14           // search for place to put current elem
15           while (moveItem > 0 && data[moveItem -
16              // shift element right one slot
17              data[moveItem] = data[moveItem - 1];
18              moveItem--;
19           }
20
21           data[moveItem] = insert; // place inser
22           printPass(data, next, moveItem); // out
23        }
24     }
25
26     // print a pass of the algorithm
27     public static void printPass(int[] data, int
28        System.out.printf("after pass %2d: ", pass
29
30        // output elements till swapped item
31        for (int i = 0; i < index; i++) {
32           System.out.printf("%d  ", data[i]);
33        }
34
```

```
      System.out.printf("%d* ", data[index]); //

      // finish outputting array
      for (int i = index + 1; i < data.length; i
         System.out.printf("%d  ", data[i]);
      }

      System.out.printf("%n  "); // for alignmen

      // indicate amount of array that’s sorted
      for (int i = 0; i <= pass; i++) {
         System.out.print("--  ");
      }
      System.out.println();
   }

   public static void main(String[] args) {
      SecureRandom generator = new SecureRandom(

      // create unordered array of 10 random int
      int[] data = generator.ints(10, 10, 91).to

      System.out.printf("Unsorted array: %s%n%n"
      insertionSort(data); // sort array
      System.out.printf("%nSorted array: %s%n",
   }
}
```

```
Unsorted array: [34, 96, 12, 87, 40, 80, 16, 50, 30,
after pass  1: 34  96* 12  87  40  80  16  50  30  45
               --  --
after pass  2: 12* 34  96  87  40  80  16  50  30  45
               --  --  --
after pass  3: 12  34  87* 96  40  80  16  50  30  45
               --  --  --  --
after pass  4: 12  34  40* 87  96  80  16  50  30  45
               --  --  --  --  --
after pass  5: 12  34  40  80* 87  96  16  50  30  45
```

```
                        --  --  --  --  --  --
   after pass  6: 12  16* 34  40  80  87  96  50  30  45
                      --  --  --  --  --  --  --
   after pass  7: 12  16  34  40  50* 80  87  96  30  45
                    --  --  --  --  --  --  --  --
   after pass  8: 12  16  30* 34  40  50  80  87  96  45
                  --  --  --  --  --  --  --  --  --
   after pass  9: 12  16  30  34  40  45* 50  80  87  96
                  --  --  --  --  --  --  --  --  --  --
   Sorted array: [12, 16, 30, 34, 40, 45, 50, 80, 87, 96
```

Fig. 19.5

Sorting an array with insertion sort.

Method `insertionSort`

Lines 8–24 declare the `insertionSort` method. Lines 10–23 loop over `data.length - 1` items in the array. In each iteration, line 11 declares and initializes variable `insert`, which holds the value of the element that will be inserted into the sorted portion of the array. Line 12 declares and initializes the variable `moveItem`, which keeps track of where to insert the element. Lines 15–19 loop to locate the correct position where the element should be inserted. The loop will terminate either when the program reaches the front of the array or when it reaches an element that’s less than the value to be inserted. Line 17 moves an element to the right in the array, and line 18 decrements the position at which to insert the next element.

After the loop ends, line 21 inserts the element into place.

Method `printPass`

The output of method `printPass` (lines 27–49) uses dashes to indicate the portion of the array that's sorted after each pass. An asterisk is placed next to the element that was inserted into place on that pass.

19.7.2 Efficiency of the Insertion Sort

The insertion sort algorithm also runs in $O(n^2)$ time. Like selection sort, the implementation of insertion sort (lines 8–24) contains two loops. The `for` loop (lines 10–23) iterates `data.length - 1` times, inserting an element into the appropriate position in the elements sorted so far. For the purposes of this application, `data.length - 1` is equivalent to $n - 1$ (as `data.length` is the size of the array). The `while` loop (lines 15–19) iterates over the preceding elements in the array. In the worst case, this `while` loop will require $n - 1$ comparisons. Each individual loop runs in $O(n)$ time. In Big O notation, nested loops mean that you must *multiply* the number of comparisons. For each iteration of an outer loop, there will be a certain number of iterations of the inner loop. In this algorithm, for each $O(n)$ iterations of the outer loop, there will be $O(n)$ iterations of the inner loop. Multiplying these values results in a Big O of $O(n^2)$.