Bubble Sort

Many sorting algorithms are available in the literature. This section describes the sorting algorithm called the **bubble sort**, to sort a list.

Suppose `list[0]...list[n - 1]` is a list of `n` elements, indexed `0` to `n - 1`. We want to rearrange, that is, sort, the elements of `list` in nondecreasing order. The bubble sort algorithm works as follows:

In a series of `n - 1` iterations, the successive elements, `list[index]` and `list[index + 1]` of `list` are compared. If `list[index]` is greater than `list[index + 1]`, then the elements `list[index]` and `list[index + 1]` are swapped, that is, interchanged.

It follows that the smaller elements move toward the top and the larger elements move toward the bottom.

In the first iteration, we consider the `list[0...n - 1]`; in the second iteration, we consider the `list[0...n - 2]` since the largest item has "sunk" to the bottom of the list; in the third iteration, we consider the `list[0...n - 3]`, and so on. For example, consider the `list[0...4]` of five elements, as shown in Figure B-1.

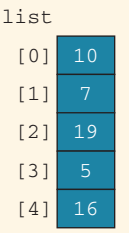

FIGURE B-1 List of five elements

Iteration 1: Sort `list[0...4]`. Figure B-2 shows how the elements of `list` get rearranged in the first iteration.

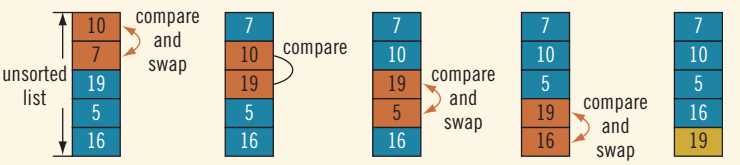

FIGURE B-2 Elements of `list` during the first iteration

Notice that in the first diagram of Figure B-2 `list[0] > list[1]`. Therefore, `list[0]` and `list[1]` are swapped. In the second diagram, `list[1]` and `list[2]` are compared. Because `list[1] < list[2]`, they do not get swapped. The third diagram of Figure B-2 compares `list[2]` with `list[3]`; because `list[2] > list[3]`, `list[2]` is swapped with `list[3]`. Then, in the fourth diagram, we compare `list[3]` with `list[4]`. Because `list[3] > list[4]`, `list[3]` and `list[4]` are swapped.

After the first iteration, the largest element is at the last position. Therefore, in the next iteration, we consider the `list[0...3]`.

Iteration 2: Sort `list[0...3]`. Figure B-3 shows how the elements of `list` get rearranged in the second iteration.

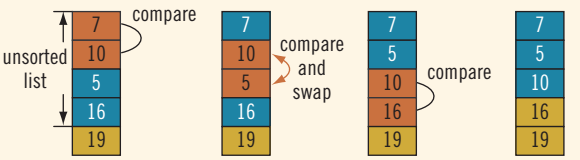

FIGURE B-3 Elements of `list` during the second iteration

The elements are compared and swapped as in the first iteration. Here, only the list elements `list[0]` through `list[3]` are considered. After the second iteration, the last two elements are in the correct places. Therefore, in the next iteration, we consider `list[0...2]`.

Iteration 3: Sort `list[0...2]`. Figure B-4 shows how the elements of `list` get rearranged in the third iteration.

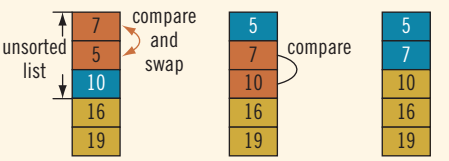

FIGURE B-4 Elements of `list` during the third iteration

After the third iteration, the last three elements are in the correct places. Therefore, in the next iteration, we consider `list[0...1]`.

Iteration 4: Sort `list[0...1]`. Figure B-5 shows how the elements of `list` get rearranged in the fourth iteration.

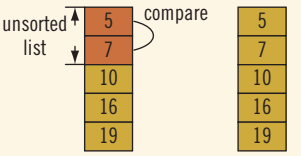

FIGURE B-5 Elements of `list` during the fourth iteration

After the fourth iteration, `list` is sorted. A smallest element is at the top of the list and does not need to be compared further against the other elements in the list.

The following method implements the bubble sort algorithm:

```
public static void bubbleSort(int list[], int listLength)
{
    int temp;
    int counter, index;

    for (counter = 0; counter < listLength - 1; counter++)
    {
        for (index = 0; index < listLength - 1 - counter;
                        index++)
            if (list[index] > list[index + 1])
            {
                temp = list[index];
                list[index] = list[index + 1];
                list[index + 1] = temp;
            }
    }
}
```

The following example illustrates how to use the bubble sort method in a program.

EXAMPLE (BUBBLE SORT)

```
public class TestBubbleSort
{
    public static void main(String[] args)
    {
        int list[] = {2, 56, 34, 25, 73, 46, 89,
                      10, 5, 16};                    //Line 1
        int i;                                       //Line 2

        bubbleSort(list, 10);                        //Line 3
```

```
        System.out.println("After sorting, the "
                           + "list elements are:");    //Line 4

        for (i = 0; i < 10; i++)                       //Line 5
            System.out.print(list[i] + " ");           //Line 6

        System.out.println();                          //Line 7
    }

    //Place the definition of the method bubbleSort
    //given previously here.
}
```

Sample Run:

```
After sorting, the list elements are:
2 5 10 16 25 34 46 56 73 89
```

The statement in Line 1 creates and initializes `list` to be an array of `10` elements of type `int`. The statement in Line 3 uses the method `bubbleSort` to sort `list`. Notice that both `list` and its length (the number of elements in it, which is `10`) are passed as parameters to the method `bubbleSort`. The `for` loop in Lines 5 and 6 outputs the elements of `list`.

For a list of length n, the bubble sort given previously makes exactly $\frac{n(n-1)}{2}$ key comparisons and on average about $\frac{n(n-1)}{4}$ item assignments. Therefore, if $n = 1000$, then to sort the list, bubble sort makes about 500,000 key comparisons and about 250,000 item assignments.

The performance of bubble sort can be improved if we stop the sorting process as soon as we find that in an iteration no swapping of elements take place. In this case, the list has been sorted.