# Visual Basic 2008 Programming

*Business Applications with a Design Perspective*

Jeffrey J. Tsay

## Table of Contents

# Appendix C

## Sorting and Searching

The main purpose of this appendix is to acquaint you with various sorting and searching algorithms, which should enable you to become more capable of dealing with new programming problems. The algorithms can provide you with new insight as to how a programming problem is solved. VB 2008 has built-in methods for sorting and searching. Therefore, the reason for discussing this topic is not its direct application but rather to help you build a foundation for programming skills.

Human beings prefer sorted data. Sorted data are easier to browse, and they facilitate lookup. Even computers have a better time with sorted data. More efficient search/processing algorithms can be used with sorted data. Thus, if an array is to be searched repetitively for different values, it pays to sort the array first. In batch processing (where transactions are collected into a batch and then all are processed at one time), transactions sorted in the same order as their master records (such as customer accounts) eliminate the need to search the master file back and forth to update the affected master records. In this case, sorting is a necessity for efficient processing.

You may be amazed that as simple as the goal of sorting is, virtually countless algorithms to sort data exist. This appendix discusses the following algorithms:

- Bubble sort
- Insertion sort
- Shell sort
- Quick sort

A careful study of the sorting algorithms allows you to appreciate the performance differences as well as helps you develop the capability to design elegant solutions to new programming problems.

A study of sorting algorithms is both important and interesting. Three decades ago, Donald E. Knuth, a well-known computer scientist asserted that nearly 20% of the computer resources at that time were used to perform sorting, according to a survey. This would suggest several alternative explanations: Sorting is important and indispensable in computer applications; sorting is not conducted efficiently; or sorting has been used unnecessarily. A careful study of sorting algorithms could help improve at least the first two situations.

Sorting entails keeping all data to be sorted in an array. Then certain algorithm of comparison and rearranging (moving) data is performed until all data are in their desired positions. The following discussion assumes that the original data have already been placed in an array.

## C.1 Bubble Sort

In the bubble sort, each element in the array is compared with the next element. If these elements are out of order, they are swapped. When one round of comparisons (and the resulting swaps) is done, the largest value in the array will be placed at the right-most position as a result. As a

result, one fewer element need be compared in the next round. This process continues until the final round in which only two elements need to be compared.
The process is illustrated in the following example.

| **Position** | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **Value** | 38 | 51 | 23 | 56 | 34 |

The comparisons start from left to right, element by element. The first element, 38, is compared with the second element, 51. Because they are in order, nothing is moved. The second element, 51, is then compared with the third element, 23. Because they are out of order, the two elements are swapped. The result appears as follows:

| **Position** | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Value | 38 | 23 | 51 | 56 | 34 |

The third element, now 51, is compared with the fourth element, 56; then 56 is compared with 34, resulting in another swap. After the first round of comparison, the array will appear as follows:

| **Position** | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Value | 38 | 23 | 51 | 34 | 56 |

Notice that the largest element in the array is now at the right-most position.

The same process of comparisons for the second round will start from position 1 again, but will stop at position 4, resulting in the second largest element being moved into this position. This process is repeated until all elements are placed in their proper position.

## *Comparing and Swapping*

How do you implement such an algorithm? Assume X() is the array to be sorted, and the Ith element is to be compared with the next element. The following statements should accomplish the comparison and swap:

```
    If X(I) > X(I+1) Then
        ' out of order; swap
        Temp = X(I)
        X(I) = X(I+1)
        X(I+1) = Temp
    End If
```

## *The Number of Elements to Compare*

What range of I should the preceding code execute? I should always start with the lower bound of the array. In the first round, it should go until I + 1 is the last element of the array; that is, I should be one less than the upper bound of the array. In the second round, this number should be one less than the preceding round; the next round, one further less, and so on. Before you have a formula to compute this value, you can name a variable, H, to represent the value. The loop to complete a round of comparisons can then appear as follows:

```
    For I = 0 To H
        If X(I) > X(I+1) Then
            ' out of order; swap
            Temp = X(I)
            X(I) = X(I+1)
            X(I+1) = Temp
        End If
    Next I
```

## The Number of Rounds of Comparisons

One round of comparisons will move only one element, the largest in the range considered, to its proper position. If the array X has n elements, it will take n – 1 rounds of comparisons to complete the sort. The sort will take n – 1, not n rounds because the smallest will automatically be placed in its position when the second smallest element is moved to its proper position. This means the preceding For loop should be executed n –1 times. In other words, the loop should be enclosed by another loop that will execute n –1 times.

```
    For J = 0  To UBound(X) - 1
        ‘ place the above For I loop here
    Next J
```

## Computing the Number of Comparisons for Each Round

You still need to compute H before the For I loop is reached. As stated previously, the first time H should have a value of one less than the upper bound of the array, and should decrease by 1 in each subsequent round. This can be done by setting its initial value before the For J loop and then decreasing it by 1 inside the J loop.

```
    H = UBound(X)
    For J = 0 To UBound(X) -1
        H -= 1
        ‘ place the above For I loop here
    Next J
```

## The Bubble Sort Sub Procedure

A sort routine is best written as a separate Sub procedure, which can then be called by any other procedure that needs a sort operation. Following this consideration, the complete Bubble Sort routine can be presented as follows:

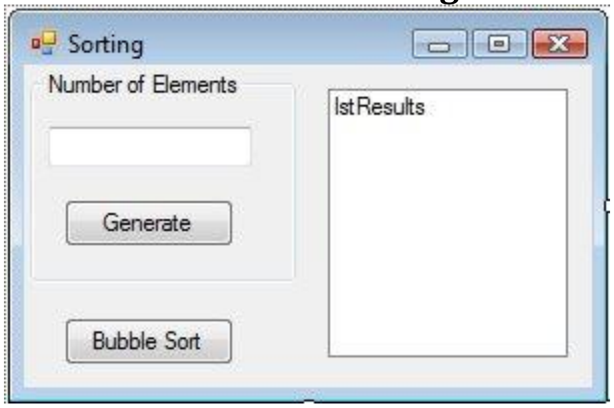
```
Sub BubbleSort(ByVal X() As Integer)
    Dim I As Integer
    Dim J As Integer
    Dim H As Integer
    Dim Temp As Integer
    H = UBound(X)
    For J = 0 To UBound(X) - 1
        H -= 1
        For I = 0 To H
            If X(I) > X(I + 1) Then
                ' out of order; swap
                Temp = X(I)
                X(I) = X(I + 1)
                X(I + 1) = Temp
            End If
        Next I
    Next J
End Sub
```

## Testing the Procedure

To test the procedure, you can design a visual interface similar to the one shown in Figure C-1. As you may surmise, the visual interface includes the following:

- A group box that contains a text box and a button. The text box is named txtNumber, and will be used to specify the number of elements for the array to be sorted. The button is named btnGenerate, and has the text "Generate." This button will be used to generate the random numbers specified in the txtNumber text box.
- Another button with the text "Bubble Sort." This button is named btnBubble, and will be used to initiate the sorting.
- A list box named lstResults. This will be used to show the sorting results.

**Figure C-1**
**Visual interface for sorting**



## Expected Behavior of the Testing Project

When the program starts, the user is expected to enter a number in the text box. When the user clicks the Generate button, your program will set the upper bound of the array X() to the number in the text box minus 1 and then generate random numbers to populate the entire array. The values of the random numbers will be in the range of 0 to this number. When the user clicks the Bubble Sort button, your program will call the Bubble Sort Sub procedure, and populate the list box with the sorting results.

## Declaring the Array

Because the array X() to hold the random numbers will be used in several procedures, it should be declared at the class level. In addition, its upper bound will change, depending on the number the user enters in the text box; therefore, it should be declared as a dynamic array. Its declaration in the form class should appear as follows:

```
    Dim X() As Integer
```

## Generating the Random Numbers

The random numbers will be generated when the user clicks the Generate button. The button's Click event procedure can appear as follows:

```
Private Sub btnGenerate_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnGenerate.Click
    Dim Number As Integer
    Dim I As Integer
    Dim Rand As New Random()
    Number = CInt(txtNumber.Text)   'Take the number from the user
    ReDim X(Number - 1) 'Set X()'s upper bound to Number
    For I = 0 To Number - 1
        X(I) = Rand.Next(Number) 'Populate X() with random numbers
    Next I
End Sub
```

In this procedure, the number entered in the text box is assigned to the variable Number; then, X() is dynamically dimensioned with its upper bound set to Number - 1. In the For loop, each element of X() is assigned a random number, in the range of 0 to Number –1 (inclusive).

## Calling the Sorting Sub and Displaying the Results

When the user clicks the Bubble Sort button, your program should call the Bubble Sort Sub procedure and then display the results. This event procedure appears as follows:

```
Private Sub btnBubble_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnBubble.Click
    Dim I As Integer
    Dim TheTime As Double
    TheTime = Microsoft.VisualBasic.Timer()
    BubbleSort(X) 'Call BubbleSort to sort the array
    Console.WriteLine(Microsoft.VisualBasic.Timer() - TheTime)
    lstResults.Items.Clear() 'Clear all previous data
    For I = 0 To UBound(X)
        'Populate the list box with sorted data
        lstResults.Items.Add(X(I))
    Next I
End Sub
```

Now, you are ready to test the Bubble Sort procedure. Run the project. Enter a number in the text box. Click the Generate button and then the Bubble Sort button. You can then inspect the numbers in the list box to verify that they are in order. (*Note:* Be sure that the list box's Sorted property is set to False, the default.)

## Computing Time Used

Notice that to see how much time the sort routine takes, two additional statements have been added. Before calling the Bubble Sort procedure, TheTime (declared as Double) is assigned a value returned by the *Timer function*, which returns a value (of the Double type) in seconds that represents the current time. (Note that the Timer function must be qualified by Microsoft.VisualBasic because of name conflict with the Timer control.) After the Sub is called, TheTime is subtracted from Timer to obtain the time it takes to perform the sorting. The result is displayed in the immediate window. Although this way of determining the time consumed is not very precise, it should give you a good idea about sorting efficiency of a particular algorithm. Notice also that the list box is first cleared with its Items.Clear method before it is populated with the sorted data. This is necessary because the user may click the Bubble Sort button several times in the duration of the program. Without that statement, the newly sorted results will be mixed with the previous ones, making it difficult to examine the results.

## A Remark on Sorting Efficiency

The speed of a sorting algorithm depends mainly on two factors: the number of comparisons and the amount of data movements. Recall that the number of comparisons involved in bubble sort varies from round to round, ranging from n – 1 in the first round to 1 in the last round; so the total is 1 + 2 + ... + (n –1) = n (n-1)/2. This number is in the order of $n^2$ and can become huge when n is large. In addition, the number of data movements can be equally huge. If you inspect the process closely, you should find that a big element located on the left side will take many swaps before reaching its final position on the right side of the array. Bubble sort is considered the least efficient sorting algorithm.

### *Simple Selection Sort*

One way to alleviate the data movement problem in the bubble sort algorithm described is to select the proper element for one position at a time; that is, instead of comparing neighboring elements in each round, the program can compare one fixed element with other elements. For example, you can start with finding the largest element in the array by comparing the last element with all other elements. If the other element is greater than this last element, the two elements are swapped. The new element in the last position is then considered the new champion until another element in the array is found to be greater. At that time, another swap will take place. This modification should reduce the number of swaps and is left to you as an exercise at the end of this appendix.

## C.2 Straight Insertion Sort

Imagine that in the sorting process, an additional array is used for output purposes. Each element from the original (input) array is placed into the output array in sorted order. When all elements from the original array are moved to the output area, the sorting is complete. To ensure that a new element from the input array is placed in the output array in sorted order, it is compared with the elements already in the output array to determine its proper position. After this position is identified, this new element is inserted in that position. This algorithm can be illustrated as follows:

The first element is moved into the output area.

| **Position** | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **Input (Original)** | 38 | 51 | 23 | 56 | 34 |
| **Output** | 38 | | | | |

The second element, 51, will then be moved to the output area. It is first compared with the existing element in output, 38. Because 51 is greater, it is placed at the end of the output area, resulting in the following:

| **Position** | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **Input (Original)** | 38 | 51 | 23 | 56 | 34 |
| **Output** | 38 | 51 | | | |

The third element will now be moved. When it is compared with the elements in the output area, you find that it should be inserted at position 1. The result will appear as follows:

| **Position** | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **Input (Original)** | 38 | 51 | 23 | 56 | 34 |
| **Output** | 23 | 38 | 51 | | |

where 23 is inserted at position 1.

This process continues until all elements in the original array are moved into the output area.

In actual implementation, there is no need to set aside another array for output purposes. Recall that the number of output elements is equal to the number of elements already moved from the input array. Because all the elements in the output area are also the same as those already moved from the input array, the area in the input array can be used to hold the output.

The algorithm involves two major activities when an element is moved from the input area to output: determining the proper position of the new element and inserting the new element in its proper position. You can write a function (for example, FindPos) to find the proper position for the new element and a Sub procedure to perform the insertion.

The *FindPos function* will search the output area sequentially until the new element is less than the element in the output or the output area is exhausted. Either of these conditions identifies the new element’s proper position. For example, if the output area has 38 and 51, and the new element is 23, the function should return the first position, as shown next:

| **Position** | 1 | 2 | 3 |
|---|---|---|---|
| **Output** | 38 | 51 | |
| | ↑ | | |

23 should be inserted here; 0 (the index for the first position) should be returned.

```
Function FindPos(ByVal X() As Integer, ByVal V As Integer, ByVal L As
  Integer) As Integer
    ' This function searches and returns the proper position
    ' for a value V in an array whose elements in the range of its
    ' lower bound and (L – 1)th position are sorted (L is imagined
    ' to be the extra position to accommodate an additional element
    ' to be inserted)
    Dim I As Integer
    For I = 0 To L - 1
        If V < X(I) Then
            Return (I)
        End If
    Next I
    Return (L)
End Function
```

## The Insert Sub Procedure

Assume P is the position in the array X() at which to insert the Lth element. In this case, all elements starting at P through L – 1 will need to be shifted to the right by one position first. The Lth element can be placed at the Pth position to complete the insertion. The steps can be depicted as follows:

**Step 1:** Move elements to the right.

| **Position** | 1 | 2 | 3 |
|---|---|---|---|
| **Output** | 38 | 51 | |
| | → | | |

**Step 2:** Insert the new element.

| **Position** | 1 | 2 | 3 |
|---|---|---|---|
| **Output** | 23 | 38 | 51 |
| | ↑ | | |

```
Sub Insert(ByVal X() As Integer, ByVal P As Integer, ByVal L As Integer)
    ' This procedure performs an insertion of X(L) at position P of
    ' Array X(); L is also the upper boundary of the sorted area
    Dim I As Integer
    Dim Temp As Integer
    Temp = X(L)  ' keep the element to be inserted
    ' Move every element in the range of p to L - 1
    ' by one position to the right
    For I = L - 1 to P Step -1
        X(I + 1) = X(I)
    Next I
    ' Insert the new element at position P
    X(P) = Temp
End Sub
```

Note that in this routine, L serves two purposes. The original element in this position is the one to be inserted. In addition, L identifies the upper boundary of the area where an insertion is to take place. Notice also that the elements are moved to the right, last elements first to avoid overwriting existing data.

## The Insertion Sort Procedure

The Insertion Sort Sub procedure can be implemented by calling these two procedures iteratively:

```
Sub InsertionSort(ByVal X() As Integer)
    Dim I As Integer
    Dim P As Integer
    For I = 1 To UBound(X)
        P = FindPos(X, X(I), I)  'Find position for X(I)
        ' Insert X(I) to position P of the sorted area
        Insert(X, P, I)
    Next I
End Sub
```

Notice that the For loop starts with 1, the second position of the array. The first element does not need to be sorted. Within the loop, the FindPos function is used to identify the proper position of X(I). The area to search for this position is between 0 and I. (The area between the lower bound and I – 1 contains the sorted elements. X(I) should be placed at position I if it is greater than all elements in this area.) After this proper position, P, is identified, the Insert Sub procedure inserts X(I) into the output area with an upper boundary set at I.

## Insertion Sort Versus Bubble Sort

When data in the array are placed randomly, this algorithm requires fewer comparisons than a bubble sort. As a new element is compared with the elements in the sorted output area, the expected number of comparisons is half of the elements in the sorted area, whereas the bubble sort algorithm requires comparisons of all possible pairs of elements. Because data movements in insertion sort involves shifting positions (to the right) rather than swapping, fewer data movements are needed than in the preceding algorithm; therefore, straight insertion sort is faster than bubble sort.

## Testing Insertion Sort

The simplest way to test this algorithm is to use the same interface and event procedures as for the bubble sort, as shown in the following steps:

1. Add the three procedures presented in this subsection (the FindPos function, the Insert sub, and the InsertionSort sub) to the same module (code window).

2. Add another button to the form. Name it **btnInsertion** and then set its Text property to **"Insertion Sort."**
3. Add the following event procedure:

```
Private Sub btnInsertion_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnInsertion.Click
    Dim I As Integer
    Dim TheTime As Double
    TheTime = Microsoft.VisualBasic.Timer()
    InsertionSort(X) 'Call InsertionSort to sort the array
    Console.WriteLine(Microsoft.VisualBasic.Timer() - TheTime)
    lstResults.Items.Clear() 'Clear all previous data
    For I = 0 To UBound(X)
        lstResults.Items.Add(X(I)) 'Add sorted element to the list box
    Next I
End Sub
```

Notice that the only difference between this procedure and the btnBubble Click procedure is the statement that calls the sort routine. Here, the Sub procedure, InsertionSort (instead of BubbleSort), is called.

To test the InsertionSort Sub, do the following:

1. Enter a number in the text box.
2. Click the Generate button.
3. Click the Insertion Sort button. (Be sure to skip the Bubble Sort button to obtain fair results.)

To obtain a truly fair comparison between the two sorting algorithms, you should populate the same random numbers in two different arrays. Each array can then be sorted by each sorting routine to obtain the sorting time. Such a modification is left to you.

### *Binary Insertion Sort*

The number of comparisons in the preceding algorithm can be further reduced if the binary search algorithm is used to search for the position for a new element. The binary search algorithm is explained in the subsection "Sequential Search and Binary Search" later in this section. Implementing this modified sorting algorithm is left to you as an exercise at the end of this chapter.

## C.3 Shell Sort

You may have noticed that in the bubble sort, each element is moved toward its proper position one swap at a time. Many swaps are usually involved before each element reaches its final position. One approach to improve this slow movement is to initially perform the comparisons of an element with another approximately half the array away. A swap of these two elements will thus accomplish a big move. The comparisons continue until no further swap is called for. This interval of comparisons can then be cut in half. The same process is repeated. Eventually, the interval will be equal to one. When it finally finishes the comparisons with this interval, you can be sure that all elements are sorted in order. (At this stage, the sort is similar to the bubble sort, but requires much fewer swaps and rounds of comparison.)

The following example illustrates how Shell sort is performed:

| | | | | | |
|---|---|---|---|---|---|
| **Position** | 1 | 2 | 3 | 4 | 5 |
| **Value** | 38 | 51 | 23 | 56 | 34 |

The initial interval is computed to be slightly smaller than half of the array; that is, 2 in this case. The first element, 38, is then compared with the third element, 23. Because the two elements are out of order, they are swapped:

| **Position** | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **Value** | 23 | 51 | 38 | 56 | 34 |

The second element, 51, is then compared with the fourth element, 56, and 38 with 34, resulting in another swap:

| **Position** | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **Value** | 23 | 51 | 34 | 56 | 38 |

The comparisons have reached the end. Because the current round resulted in swaps, another round of comparisons with the same interval should be performed; that is, 23 with 34, 51 with 56, and 34 with 38. No swap occurs. So, the interval is reduced by half to 1, and another round of comparisons is performed until no swap occurs.

Assume X() is the array to be sorted. You can first compute the interval as follows:

```
    Interval = CInt(Math.Ceiling(UBound(X) / 2))
```

Does the formula appear more complex than necessary? Actually, it is written this way to take care of a fine point. The reason will be explained shortly.

A For loop as shown here should complete a round of comparisons:

```
    For I = 0 to UBound(X) - Interval
        If X(I) > X(I + Interval) Then
            ' Out of order, swap
            Temp = X(I)
            X(I) = X(I + Interval)
            X(I + Interval) = Temp
        End If
    Next I
```

The For statement sets the position of the element on the left to be compared and starts at the lower bound of the array. It should end when I + Interval is greater than the upper bound of X; that is, I should go as high as I + Interval = UBound(X). Thus, the ending value for I is Ubound(X) – Interval.

This loop is to be repeated until no swap occurs. To keep track of swaps, you can use a Boolean variable, Swapped. The variable can be set to False before executing the For loop. If a swap occurs, it is set to True. Before the outer loop is repeated, this variable can be tested, and another round of comparisons is performed only if Swapped is True.

```
    Do
        Swapped = False
        For I = 0 to UBound(X) - Interval
            If X(I) > X(I + Interval) Then
                ' Out of order, swap
                Temp = X(I)
                X(I) = X(I + Interval)
                X(I + Interval) = Temp
                Swapped = True
            End If
        Next I
    Loop While Swapped
```

When the outer loop is finished, it is time to reduce the interval size by half and then repeat another round of comparisons until the interval size is zero. The entire sort procedure using the Shell sort algorithm should appear as follows:

```
Sub ShellSort(ByVal X() As Integer)
    Dim Interval As Integer
    Dim I As Integer
    Dim Swapped As Boolean
    Dim Temp As Integer
    Interval = CInt(Math.Ceiling(UBound(X) / 2))
    Do Until Interval = 0
        Do
            Swapped = False  ' assume no swap
            For I = 0 To UBound(X) - Interval
                If X(I) > X(I + Interval) Then
                    ' Out of order, swap
                    Temp = X(I)
                    X(I) = X(I + Interval)
                    X(I + Interval) = Temp
                    ' Swap occurs, so set swapped to true
                    Swapped = True
                End If
            Next I
        Loop While Swapped
        ' Reduce the interval by half
        Interval = CInt(Interval / 2)
    Loop
End Sub
```

Notice that two different formulas are used to compute Interval. The second formula computes Interval by converting into the integer value the quotient of the previous Interval divided by 2. CInt rounds its parameter; however, when the fractional portion is exactly .5, it rounds the parameter to the nearest even number. That is, if the parameter is 1.5, it will be rounded to 2; but if the parameter is .5, it will be rounded to 0. This is exactly what you want for it to do in the second formula. But consider the formula at the beginning. If X has two elements (thus its UBound is 1), Interval will be 0 if you use the same formula as the second one. This will mean that when X has two elements, the array will never be sorted. To ensure that Interval will be at least one, the Ceiling function, which gives the smallest integer greater than or equal to the parameter, is first used to round the number up before the result is converted to an integer by the CInt function.

You can implement a test procedure by following the same steps as outlined in the preceding section. Test the program, and you should find that this algorithm is much faster than those previously discussed.

## C.4 Quick Sort

You may recall that in the bubble sort, in each round of comparisons, the algorithm attempts to identify which element in the array should be moved to a destination position. In other words, a destination is waiting for an element that qualifies. In the quick sort, the goal is inverted. The algorithm looks for the proper position for the given data element at hand. Specifically, when the sorting operation starts, the first element in the array is considered the pivotal element. The algorithm looks for the proper final position in the array to place this pivotal element. This is done by first comparing the pivotal element with the array elements backward (right to left) until the pivotal element is greater than the element in the array. The pivotal element is swapped with this element; then the pivotal element (in its new position) is compared with elements in the array forward (left to right) until it is found to be smaller than another element in the array. A swap takes place, and the comparisons go backward again.

The backward-forward sequence of comparisons/swaps continues until all elements in the array have been compared with the pivotal element. At this point, no element on the left side of the pivotal element is greater than and no element on its right is less than the pivotal element. The pivotal element has found its proper position. This process also ensures that all elements on the left side are smaller than all elements on the right side of the pivotal element; no further sorting between the two sides is necessary. The elements on each side can then be sorted separately using the same algorithm.
The following example illustrates how the pivotal element is placed in its proper position.

| **Position** | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **Value** | **38** | 51 | 23 | 56 | 34 |

$\xleftarrow{\qquad\qquad\qquad}$

The first element, 38, is the pivotal element. At first, the comparisons start from right to left until the pivotal element is greater than the element in the array. This occurs immediately when 38 is compared with the fifth element, 34. A swap takes place:

| **Position** | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **Value** | 34 | 51 | 23 | 56 | 38 |

$\xrightarrow{\qquad\qquad\qquad\qquad}$

Comparisons now turn forward. (Notice that 38 is the pivotal element.) The second element, 51, is compared with 38. They are out of order. Another swap takes place:

| **Position** | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **Value** | 34 | 38 | 23 | 56 | 51 |

$\xleftarrow{\qquad\qquad\qquad}$

Comparisons then go backward again. 38 is compared with 56 and then with 23, where another swap takes place:

| **Position** | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **Value** | 34 | 23 | 38 | 56 | 51 |

At this point, all elements have been compared with 38. The pivotal element has found its proper position. Note that all elements on its left side are smaller than 38, and all elements on its right side are greater. The sub-arrays on both sides (as depicted in the following illustration) can be sorted by the same algorithm:

| **Position** | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **Value** | ( 34 | 23 ) | 38 $\uparrow$ | ( 56 | 51 ) |

## *Header for Quick Sort*

Before you start coding for this algorithm, you should note that you will use the same procedure to sort sub-arrays. This means that this sort Sub procedure will have to call itself; that is, the sub will be used as a recursive procedure. Each sub-array has a different range of elements to sort. The range can be defined by the beginning and ending positions of the sub-array; therefore, the header of this Sub procedure should look a bit different from the previous sort procedures:

```
Sub QuickSort(ByVal X() as Integer, ByVal BegPos as Integer, ByVal EndPos as Integer)
```

where X() is the array to be sorted, and BegPos and EndPos define the range of the sub-array to sort. Of course, when the routine is initially called, these two variables will cover the entire range—0 and UBound(X)—of the array.

You will use two variables—I and J—to keep track of the positions for forward and backward comparisons, respectively. Initially, I starts at the beginning of the range, and J starts at the end:

```
    I = BegPos
    J = EndPos
```

The first element is the pivotal element:

```
    Pivot = X(I)
```

## Comparing Backward

The comparisons will first go backward. For any element J in the array, if it is found to be greater than or equal to Pivot, you will go one position to the left; otherwise, a swap takes place, and the backward comparison ends:

```
    If Pivot =< X(J) Then
        ‘ In order, go one position left
        J = J - 1
    Else
        ‘ out of order; swap and terminate the loop
        X(I) = X (J)
        X(J)= Pivot
        Exit Do
    End If
```

Note that before the swap, X(I) and Pivot are the same. There is no need to assign X(I) to Pivot in the swap process. This comparison routine should be enclosed in a Do loop. If the comparisons exhaust all the items (that is, I = J), the loop should also end. The complete code for backward comparisons should appear as follows:

```
    Do Until I >= J
        If Pivot <= X(J) Then
            ‘ In order, go one position to the left
            J = J - 1
        Else
            ‘ out of order; swap and terminate the loop
            X(I) = X (J)
            X(J)= Pivot
            Exit Do
        End If
    Loop
```

## Comparing Forward

After the preceding loop ends, the process should proceed to the next phase to compare forward. The code can be written in a similar fashion as follows:

```
    Do Until I >= J
        If Pivot >= X(I) Then
            ‘ In order, go one position to the right
            I = I + 1
        Else
            ‘ out of order; swap and terminate the loop
            X(J) = X (I)
            X(I)= Pivot
            Exit Do
        End If
    Loop
```

Again, notice that at this point before the swap, Pivot and X(J) are the same (see the swap in the preceding loop). There is no need to assign X(J) to Pivot at the beginning of the swap.

## The Outer Loop

The preceding two loops should be repeated as long as some elements still have not been compared with Pivot; that is, another (outer) loop as shown below should enclose the two preceding loops:

```
    Do Until I >= J
        ' Place both of the above loops here
    Loop
```

## Sorting the Sub-arrays

The preceding code completes the process of placing the pivotal element in its proper position. As stated at the beginning of this section, after this is done, the sub-arrays on both sides of the pivotal element can be sorted using the same algorithm. The following two lines of code should be placed immediately after the preceding loops:

```
    QuickSort(X, BegPos, J - 1) 'Sort the subarray on left side
    QuickSort(X, J + 1, EndPos) ' sort the subarray on right side
```

## Terminating the Procedure

The procedure so written will continue to call itself unless you provide additional code to end the recursive process. The procedure should discontinue further sorting under the following circumstances:

- There is only one element in a sub-array; therefore, there will be no need to sort.
- There are only two elements in a sub-array, so you can perform a simple comparison to see whether they are out of order. If so, a swap can be performed; otherwise, nothing further needs to be done. In either case, there is no need to perform additional sorting.

This analysis suggests the following code:

```
    If EndPos - BegPos < = 0 Then
        Exit Sub 'One or no elements, no need to sort
    End If
    If EndPos - BegPos = 1 Then
        ' This subarray has two elements, check to see if they are
        ' out of order
        If X(BegPos) > X(EndPos) Then
            ' Out of order, swap
            Temp = X(BegPos)
            X(BegPos) = X(EndPos)
            X(EndPos) = Temp
        End If
        Exit Sub
    End If
```

These lines should be placed at the very beginning of the procedure.

## The Better, the Worse

The code previously presented basically completes the quick sort algorithm in its original form. When used to sort an array with random elements, it is fast because this method preserves the results of previous comparisons and involves few unnecessary data movements. Recall that all elements found to be smaller than the pivotal element are placed on its left, and all elements greater are on its right. Therefore, none of the elements on the left need to be compared with those on the right, eliminating many unnecessary comparisons that occur in the bubble sort. The process of searching for the proper position of the pivotal element also moves each element near its proper position. When used to sort an array that is already in order, however, the number of comparisons degenerates to the bubble sort because at the end of each round, the pivotal element is placed at the beginning of the array, resulting in only one side to be sorted. The next pivotal element will again be compared with the remainder of the entire array.

## Solving the Paradox

One proposal to alleviate this problem is to select a random element (instead of the first element) as the pivotal element. Another proposal is to use the midpoint of the array as the pivotal element. If you use the midpoint approach, you will first swap the midpoint element with the first element and then proceed with the sorting as previously presented. That is, you will add the following statements:

```
    M = CInt((BegPos + EndPos) / 2 ) 'Find mid point
    ' Use mid point element as the pivotal element
    ' Also swap the mid point element with the first element
    Pivot = X(M)
    X(M) = X(BegPos)
    X(BegPos)= Pivot
```

These statements will replace the line Pivot = X(I) presented previously.

## The Stack Space Issue

Another issue concerning quick sort is the amount of stack space needed. The Sub procedure calls itself when sorting the sub-arrays on both sides of the pivotal element. The parameters passed to the procedure itself are kept in the memory stack. If not carefully managed, the routine can run out of stack space before the sort routine is complete. The solution is to ensure that the shorter sub-array is sorted first. To implement this solution, the calls to sort the subarrays can be rewritten as follows:

```
    If J - BegPos <= EndPos - J Then
        ' Left side is shorter, sort it first
        QuickSort(X, BegPos, J - 1)  'Sort the left subarray
        QuickSort(X, J + 1, EndPos) ' sort the right subarray
    Else
        ' Right side is shorter, sort it first
        QuickSort(X, J + 1, EndPos) ' sort the right subarray
        QuickSort(X, BegPos, J - 1)  'Sort the left subarray
    End If
```

## The Complete Code for Quick Sort

The complete Quick Sort routine as modified appears as follows:

```
Sub QuickSort(ByVal X() As Integer, ByVal BegPos As Integer, ByVal EndPos As
Integer)
    Dim Temp As Integer
    Dim Pivot As Integer
    Dim M As Integer
    Dim I As Integer
    Dim J As Integer
    If EndPos - BegPos <= 0 Then Exit Sub 'One or fewer elements,
       ' no need to sort
       If EndPos - BegPos = 1 Then
           ' This subarray has two elements.
           ' Check whether they are out of order
           If X(BegPos) > X(EndPos) Then
               ' Out of order, swap
               Temp = X(BegPos)
               X(BegPos) = X(EndPos)
               X(EndPos) = Temp
           End If
           Exit Sub 'No need to do anything more
       End If
       ' Here is the typical quick sort
       I = BegPos   ' set initial left position for forward comparisons
       J = EndPos   ' set initial right position for backward comparisons
       M = CInt((BegPos + EndPos) / 2)  'Midpoint of array
       Pivot = X(M) ' Use midpoint of array as pivotal element
       X(M) = X(BegPos) ' Swap midpoint with first element
       X(BegPos) = Pivot
       ' Find the proper position for Pivot
       Do Until I >= J
           ' Check backward
           Do Until I >= J
               If Pivot <= X(J) Then
                   ' In order, go one position to the left
                   J = J - 1
               Else
                   ' out of order; swap and terminate the loop
                   X(I) = X(J)
                   X(J) = Pivot
                   Exit Do
               End If
           Loop
           ' Check forward
           Do Until I >= J
               If Pivot >= X(I) Then
                   ' In order, go one position to the right
                   I = I + 1
               Else
                   ' out of order; swap and terminate the loop
                   X(J) = X(I)
                   X(I) = Pivot
                   Exit Do
               End If
           Loop
       Loop
       ' Sort subarrays
       If J - BegPos <= EndPos - J Then
           ' Left side is shorter, sort it first
           QuickSort(X, BegPos, J - 1)   'Sort the left subarray
           QuickSort(X, J + 1, EndPos) ' sort the right subarray
       Else
           ' Right side is shorter, sort it first
           QuickSort(X, J + 1, EndPos) ' sort the right subarray
           QuickSort(X, BegPos, J - 1)   'Sort the left subarray
```

```
        End If
End Sub
```

This completes the discussion of the quick sort algorithm. As a reminder, the code to call this sort routine is a bit different, and should appear as follows:

```
    QuickSort(X, 0, UBound(X))
```

The next section presents sample empirical performance results of various sorting algorithms. You should find the quick sort algorithm gets its name for a good reason.

## C.5 Comparison of Performance

As you can see, some of the algorithms are pretty simple; but others are more involved. In general, if two algorithms give the same performance, you would rather use the simpler one. The additional complexity can be justified only with better performance. So, how do they compare in terms of speed? The following table shows the time used to sort varying number of elements by these algorithms using a Pentium IV 1.4 GHz machine.

| Number of Elements | Bubble Sort | Insertion Sort | Shell Sort | Quick Sort |
|---|---|---|---|---|
| 4000 | 0.12017 | 0.05007 | 0 | 0 |
| 8000 | 0.48069 | 0.17024 | 0.01001 | 0 |
| 16000 | 1.92276 | 0.67096 | 0.03004 | 0 |
| 32000 | 7.68104 | 2.61376 | 0.07010 | 0.01001 |
| 64000 | 30.87440 | 10.63529 | 0.18026 | 0.02003 |

The leftmost column shows the number of integer numbers (generated by the formula given in bubble sort) being sorted. The numbers in the table are time in seconds. Because the numbers are randomly generated, the results can be different if you attempt to replicate the experiments; however, you should be able to make several general observations:

- Quick sort and Shell sort far outperform the other algorithms.
- As the number of elements to sort doubles, the first two algorithms approximately quadruple the time used; the latter two algorithms slightly more than double.

The nearly linear relationship between the number of elements and time required to sort suggests that the last two algorithms are not only more efficient, but also steadier performers.

## C.6 Sequential Search and Binary Search

After data in an array are sorted in order, they can be searched with more efficient algorithms. Data stored randomly can be searched only sequentially. The average number of comparisons required to find an item will be half of the number of elements in the array.

### *Improved Sequential Search with Sorted Data*

The number of comparisons can be reduced if the data are arranged in order, even with the same sequential search method. How? Because the data are in order, if the data in the array is found to be greater than the search key, there is no need to search further. All data beyond this point will be greater than the search key. The search can therefore be terminated with the conclusion, “Data Not Found.” The code to implement this algorithm is left you.

## *The Binary Search*

The search can be even more efficient if the binary search algorithm is used. This algorithm begins by setting the lower and upper boundary of the search to the entire range of the array. It then starts the search at the midpoint of the array. If the search key is greater than the data at the midpoint, the item being searched (if it exists) must be in the upper half of the array; the lower boundary can be adjusted to this midpoint. On the other hand, if the search key is less than the element in the array, the item being search must be in the lower half; therefore, the upper bound is adjusted. The midpoint of this new search boundary is then computed. The search continues, each time narrowing the search boundary by half until the item is found or there is only one item in the range (lower and upper bounds are the same; this means data for the search key does not exist). This algorithm is similar to the half interval method introduced in Chapter 7, “Repetition,” to find a numerical solution for a mathematical function. (See Figure 7-4 for a sketch of the algorithm.) You can implement this algorithm to search for a value in a string array with the following code:

```
Private Function BinSearch(X() As String, SearchKey As String) As Integer
    Dim Name As String
    Dim I As Integer
    Dim Lower As Integer
    Dim Upper As Integer

    Lower = 0
    Upper = X.Length -1
    Do
        I = (Lower + Upper) \ 2 'Compute the mid point
        If SearchKey = X(I) Then
            Return(I)
        ElseIf SearchKey > X(I) Then
            ' the value in array is less than the search key;
            ' Adjust the lower bound
             Lower = I + 1
        Else
            ' The value in array is greater than the search key;
            ' Adjust the upper bound
             Upper = I - 1
        End If
    Loop Until Lower = Upper
    If SearchKey = X(Upper) Then
       Return(Upper)
    Else
        Return(-1)
    End If
End Sub
```

The function will return the position in which the search key is found. If there is no matching value, the function returns -1. Notice that after the Do loop, SearchKey is compared with X(Upper) to determine for the last time that the SearchKey is actually in the array. This is necessary because when Lower and Upper reach the same value, a new I value has not been computed.

Before leaving this example, note that the binary search method is commonly used. VB 2008 provides a BinarySearch method for the Array object. The purpose here is to show you the algorithm itself, and hope you will be able use it to solve other similar problems.

## Summary

- The basic idea of bubble sort to is compare two neighboring values in an array and swap them if they are out of order. The process gradually place items in their properly position beginning with the greatest value in the array being placed in the highest position. Iteratively apply the same process will eventually arrange all elements in the arrays in order. This method is the least efficient.
- Insertion sort picks up an element from an input array (from the lowest to the highest position) and insert it in the sorted output array in the position such that the sorted order is preserved. This method is slightly more efficient than bubble sort.
- Shell sort starts with a comparison (and swap) interval of one half of the array. Rounds of comparisons with this interval are repetitively performed until no more swaps are required. Then the comparison interval is halved. This process is repeated until the interval is one and no more swaps are required. Shell sort is much more efficient than the preceding two methods.
- Quick sort seeks to find the proper position for a pivotal element. After reaching its final destination, all elements on its left are smaller and on its right are greater than this pivotal element. The same process can be used to sort elements on both sides the pivotal element. When properly implemented, quick sort is the most efficient method among the four algorithms described in this appendix.

### Exercises

**C-1. A Possible Improvement on the Bubble Sort.** A suggested method to improve the performance of the bubble sort is to add a counter in the inner loop. The counter will count the number of swaps. If there is no swap after the loop is complete, the counter will be zero. This will be an indication that all the elements in the array are already in order; therefore, the process can be terminated immediately. The counter adds overhead to the sorting process, but allows the algorithm to perform faster when the array to be sorted is already pretty much in order. Modify the code in this appendix to implement this improvement.

**C-2. Simple Selection Sort.** Write a Sub procedure to perform the simple selection sort as described in Section C-1.

**C-3. Reverse Simple Selection Sort.** The simple selection sort as described in Section C-1 has an alternative. Instead of selecting the largest element first, you can start with selecting the smallest element first and placing it in the first position of the array. You can then proceed to find the second, third,... and $n^{th}$ smallest elements in the same manner. Modify the program as required in exercise C-2 to implement this alternative sorting algorithm for simple selection.

**C-4. Binary Insertion Sort.** Write a Sub procedure to perform the binary insertion sort as described in Section C-2.

**C-5. Improved Shell Sort.** Empirical studies have found that Shell sort is most efficient when the intervals of sort are computed as follows:

```
Interval 1 = 1
Interval k + 1 = 3 x Interval k + 1
And stops when Interval k + 2 > Number of elements to be sorted
```
That is, the intervals can be computed as follows:

| K Value | Interval |
|---|---|
| 1 | 1 |
| 2 | 3 x 1 + 1 = 4 |
| 3 | 3 x 4 + 1 =13 |
| 4 | 3 x 13 + 1 = 40 |
| 5 | 3 x 40 + 1 =121 |
| 6 | 3 x 121 + 1 = 364 |

An array with 300 elements should be sorted with the sort intervals set to 40, 13, 4, and 1 according to this table because two steps below it, the interval 364 is greater than the number of elements 300. Modify the Shell sort routine as presented in Section C-3 to implement this improvement.

**C-5. Reverse Quick Sort.** You may have noticed that in the quick sort algorithm, if you use the last element in the array instead of the first element as the pivotal element, the comparisons should start forward first until a swap is called for; then followed by backward comparisons. The backward/forward comparison sequence in search for the proper position for the pivotal element as presented in Section C-4 is changed to a forward/backward sequence. Just for the fun of it, revise the quick sort procedure to implement this modification.