Speed Sorting with Seven Algorithms

Want to speed up your VB sorting tasks? Here's how to choose and use sorting algorithms for faster results.

BY DEEPAK AGRAWAL

There's no one best sorting algorithm for every task. Of the many algorithms you can choose, some are dramatically more efficient than others. Some are easier to write and use than others. Still others may be extremely fast, but pose resource problems and crash if not properly thought out.

In order to determine which sort is right for you, you need to know the capabilities of each sort method, the complexity of the sort and the process by which the sort operates. Let's look at seven sorts: Insertion, Selection, Bubble, Bucket, Shell, Heap, and Quick sort.

Each sorting algorithm falls into one of two categories based on its runtime execution efficiency: Insertion, Selection, Bucket, and Bubble sort are all N ** 2; Merge, Heap and Quick sort are N log N (log base 2); the Shell sort can belong to either group.

If N is assigned to be indicative of the size of the array, then you can watch an N ** 2 sort to see how much time is consumed during execution. The larger the size of N, the longer the sort takes to execute. If our N array is 100 elements, it could take 10,000 iterations to complete the sort. If you were to sort an array of 15,000 user IDs from a Customer table, this could take as long as 225,000,000 iterations to complete. With the N log N sorts this would be substantially faster: as small as 62,641 iterations for the same array. That is roughly 99.98 percent faster. It sounds good, but these sorts are far more complex.

Each sort routine accepts an array of random data, internally sorts the elements of the array and eventually returns the array in properly sorted order. The pseudocode logic for each sort is included so that you can re-create the routines in the language of your choice. The pseudocode is used to show the process of the sorts, and won't run in VB. You must translate the pseudocode examples here into VB or C++ code. An example program uploaded to the *VBPJ* Forum on CompuServe and printed here (see Listing 1) will give you VB code using these techniques and will let you evaluate the different sorts more fully. Look for VBSORT.ZIP in the Magazine Library (see Figure 2).

Each of these sorts has been implemented in straight VB code, without the use of any DLL libraries, and are located in the SORT.BAS file. Some recursive and nonrecursive implementations will be demonstrated. Bone up on your understanding of recursive routines. Not many VB programs take advantage of this optimization technique (see "The Secrets of Recursion" [*VBPJ* December/January 1993/1994] for more on recursive routines, and "C DLL Extends VB Array Sorting" [Windows Programming, *VBPJ* February/March 1994] for more on the sorting DLL).

The pseudocode logic for the sorts uses these variables:

V: To represent the array passed to the sorting function.
L: Pointer to Low entry in the array (the Lbound value).
H: Pointer to the High entry in the array (the Ubound value).
N: Size of the array.
I, J, K: Temporary variable pointers (Indexes into the array).

Let's take a look at the Bubble sort and what makes it tick. Classic sorting algorithms take advantage of pointers, but because we are VB developers and do not yet have the luxury of pointers, these sorting routines have been altered to fit the needs of the Visual Basic programmer.

The Bubble sort is like the tortoise from Aesop's fable about the tortoise and the hare: it's the underdog (see Figure 1). While the Bubble sort is by far the slowest, most inefficient algorithm for large jobs, it is also the easiest to code. That's what makes it so popular with programmers.

The Bubble sort operates on the principle that eventually the smallest number will bubble to the top or bottom of the array.

The Bubble sort is inefficient due to the method by which it determines the relationship of the elements in the array to be sorted. A comparison of adjacent elements is made at every stage of the sorting process. The larger the array, the larger the

Deepak Agrawal is president of DA Consulting Inc., a Chicago, Illinois-based consultancy specializing in client/server application development, training, downsizing corporate systems, and providing progressive technology services. He also contributes to a variety of technical publications and industry newsletters, and is a guest conference speaker. Deepak can be reached at 708-742-9985, or on CompuServe at 73322,1561.

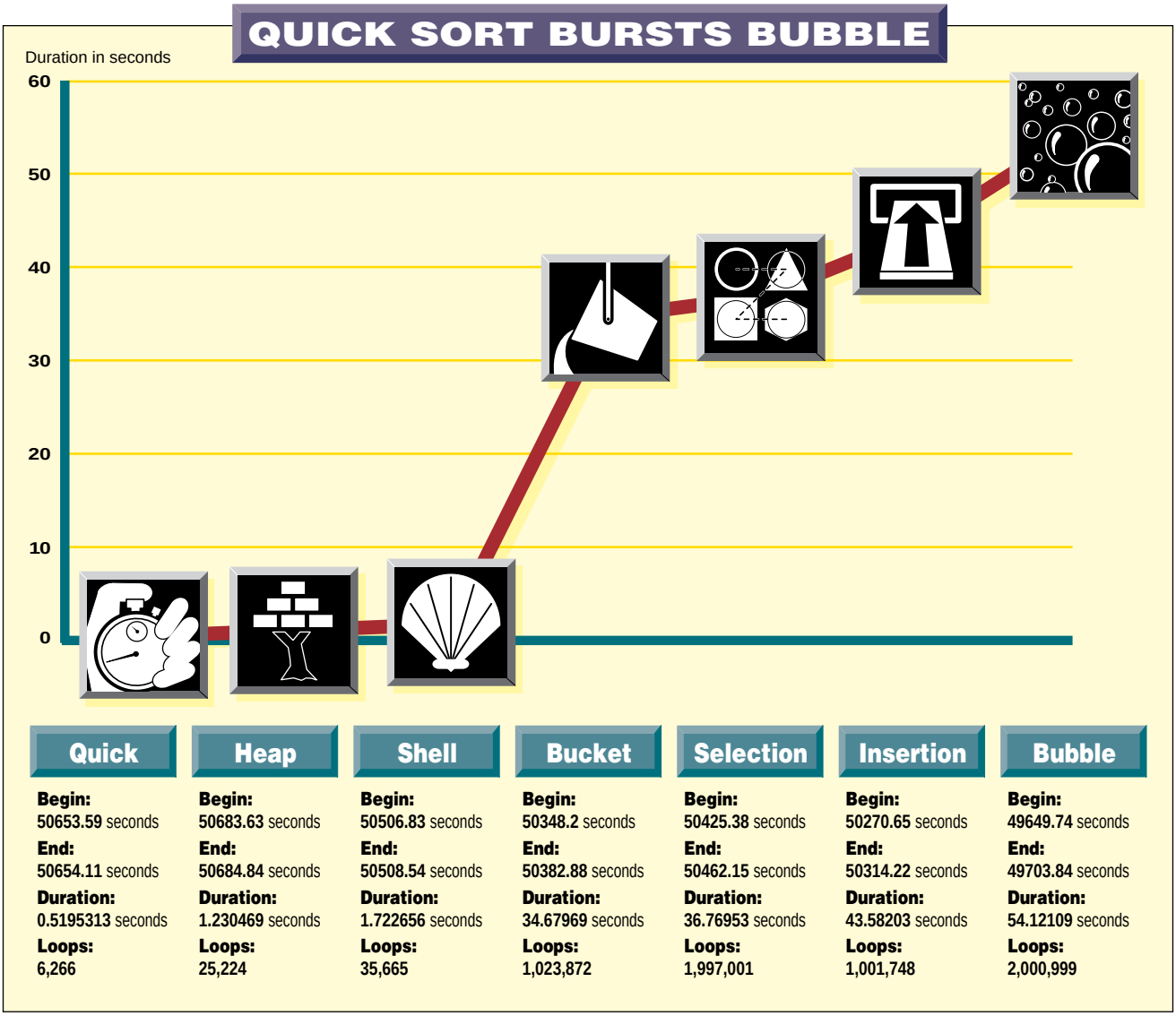

FIGURE 1 ***Going Around in Circles.*** *Even on a 90 Mhz Pentium with a math-coprocessor, Windows NT 3.5, and 32 MB of RAM, the Bubble sort will make you want to upgrade your machine (these timings were made on such a system). Keep in mind that these timings will vary on your machine, but the percent differences will be consistent. The loop count is only an indication of how many iterations were performed, not how many comparisons were made.*

number of comparisons and the longer it will take the sort to complete. One unpopular characteristic of the Bubble sort is that if a sorted array is passed to it, it will still try to sort the array.

Here is the pseudocode for a typical Bubble sort:

```
Sub Bubble (V, L, H )

    J = L
    DO WHILE ( J < H )
        K = H
        DO WHILE ( K > J )
            IF V( K ) < V( K - 1 ) _
                THEN
            TEMP = V( K )
                V( K ) = V( K - 1 )
                V( K - 1 ) = TEMP
                ENDIF
                K = K - 1
            LOOP
                'At this point the
                'lowest element is
                'at the top
            J = J + 1
    LOOP
End Sub
```

The Bubble sort compares each element until the array is sorted in ascending order (see Figure 3). With this set of data the Bubble sort would take this step-by-step sequence:

```
point J to element 0
point K to element 2
compare element K (7) to element K-1 _
    (12) : A
swap 7 and 12
decrement K by 1
compare element K (7) to element K-1 _
    (-5): B
don't swap because 7 > -5, so _
    decrement K by 1
since K is not > J exit inner loop _
    and begin again : C
point K to element 2, point J to _
    element 1
compare element K (12) to element K-1 _
    (7) : D
don't swap because 12 > 7, so _
    decrement K by 1
since K is not > J exit inner loop _
    and begin again : E
point J to element 2 : F
```

CONTINUED ON PAGE 44.

```
SORT.BAS
Global Const ZERO = 0
Global Const ASCENDING_ORDER = 0
Global Const DESCENDING_ORDER = 1

Global gIterations

'***BUBBLE***
Sub BubbleSort (Array(), ByVal
nOrder As Integer)
Dim Index
Dim TEMP
Dim NextElement

  NextElement = ZERO
  Do While (NextElement < _
    UBound(Array))
  Index = UBound(Array)
  Do While (Index > NextElement)
    If nOrder = ASCENDING_ORDER _
      Then
    If Array(Index) < Array(Index _
    1) Then
      TEMP = Array(Index)
      Array(Index) = Array(Index -1)
      Array(Index - 1) = TEMP
    End If
    ElseIf nOrder = _
      DESCENDING_ORDER Then
    If Array(Index) >= Array(Index_
    1) Then
      TEMP = Array(Index)
      Array(Index) = Array(Index - _
        1)
      Array(Index - 1) = TEMP
    End If
    End If
    Index = Index - 1
    gIterations = gIterations + 1
  Loop
  NextElement = NextElement + 1
  gIterations = gIterations + 1
  Loop

End Sub

'***BUCKET***
Sub Bucket (Array(), ByVal nOrder
As Integer)
Dim Index
Dim NextElement
Dim TheBucket

  NextElement = LBound(Array) + 1
  While (NextElement <= _
    UBound(Array))
  TheBucket = Array(NextElement)
  Index = NextElement
  Do
    If Index > LBound(Array) Then
    If nOrder = ASCENDING_ORDER _
      Then
      If TheBucket < Array(Index _
        - 1) Then
      Array(Index) = Array(Index _
        - 1)
      Index = Index - 1
      Else
      Exit Do
      End If
    ElseIf nOrder = _
      DESCENDING_ORDER Then
      If TheBucket >= Array(Index _
        - 1) Then
      Array(Index) = Array(Index - _
        1)
      Index = Index - 1
      Else
      Exit Do
      End If
    End If
    Else
    Exit Do
    End If
    gIterations = gIterations + 1
  Loop
  Array(Index) = TheBucket
  NextElement = NextElement + 1
  gIterations = gIterations + 1
  Wend

End Sub

'***HEAP***
Sub Heap (Array())
Dim Index
Dim Size
Dim TEMP
  Size = UBound(Array)

  Index = 1
  While (Index <= Size)
  Call HeapSiftup(Array(), Index)
  Index = Index + 1
  gIterations = gIterations + 1
  Wend

  Index = Size
  While (Index > 0)
  TEMP = Array(0)
  Array(0) = Array(Index)
  Array(Index) = TEMP
  Call HeapSiftdown(Array(), _
    Index - 1)
  Index = Index - 1
  gIterations = gIterations + 1
  Wend

End Sub


Sub HeapSiftdown (Array(), M)
Dim Index
Dim Parent
Dim TEMP

  Index = 0
  Parent = 2 * Index

  Do While (Parent <= M)

  If (Parent < M And Array(Parent) _
    < Array(Parent + 1)) Then
    Parent = Parent + 1
  End If

  If Array(Index) >= Array(Parent) _
    Then
    Exit Do
  End If

  TEMP = Array(Index)
  Array(Index) = Array(Parent)
  Array(Parent) = TEMP

  Index = Parent
  Parent = 2 * Index

  gIterations = gIterations + 1
  Loop

End Sub

Sub HeapSiftup (Array(), M)
Dim Index
Dim Parent
Dim TEMP

  Index = M
  Do While (Index > 0)
  Parent = Int(Index / 2)

  If Array(Parent) >= _
    Array(Index) Then
    Exit Do
  End If

  TEMP = Array(Index)
  Array(Index) = Array(Parent)
  Array(Parent) = TEMP

  Index = Parent
  gIterations = gIterations + 1
  Loop

End Sub

'***INSERTION***
Sub Insertion (Array(), ByVal _
  nOrder As Integer)
Dim Index
Dim TEMP
Dim NextElement

  NextElement = LBound(Array) + 1
  While (NextElement <= _
    UBound(Array))
  Index = NextElement
  Do
    If Index > LBound(Array) Then
    If nOrder = ASCENDING_ORDER _
      Then
      If Array(Index) < _
        Array(Index - 1) Then
      TEMP = Array(Index)
      Array(Index) = _
        Array(Index - 1)
      Array(Index - 1) = TEMP
      Index = Index - 1
      Else
      Exit Do
      End If
    ElseIf nOrder = _
      DESCENDING_ORDER Then
      If Array(Index) >= _
        Array(Index - 1) Then
      TEMP = Array(Index)
      Array(Index) = _
        Array(Index - 1)
      Array(Index - 1) = TEMP
      Index = Index - 1
      Else
      Exit Do
      End If
    End If
    Else
    Exit Do
    End If
    gIterations = gIterations + 1
  Loop
  NextElement = NextElement + 1
  gIterations = gIterations + 1
  Wend

End Sub
```

CONTINUED ON PAGE 44.

LISTING 1 ***Sort Out Your Life.*** *Here's the code for seven sorting algorithms: Insertion; Selection; Bubble; Bucket; Shell; Heap; and Quick sort. Some are easier to code, while others will be radically faster at run time (see Figure 1).*

CONTINUED FROM PAGE 39.

```
since J not < H exit loop
Array Is Sorted
```

The number of interactions performed by the Bubble sort varies depending on the size and randomness of the array. In this example, the sort performed five tests before the array was sorted properly. Because the worst-case scenario would be nine tests (N ** 2), for a small set of data, the Bubble sort does an adequate job of sorting in a reasonably short period of time.

FROM BUBBLE TO INSERTION

The basic idea in an Insertion sort is much like what card players use to arrange cards. You look at each card one at a time and when each new card is seen, it is inserted into its proper place.

At each stage of the Insertion sort, the cards preceding the current card are already sorted. Rather than run through the entire list of members, you compare the current card with the partial set of sorted members. The farther we go to the bottom of the deck the longer the sort takes because there are more members to test. Compared to the Bubble sort, which tests every member in the array every time, the Insertion sort actually begins with a small comparison and increases gradually.

The Insertion sort example's logic is very similar to the Bubble sort. You begin by comparing adjacent elements and swapping them if they are not in the proper ascending or descending order. The efficiency of the Insertion sort comes from its ability to sort a small set of members before continuing through the remaining unsorted members of the array, until the entire array is sorted.

This pseudocode for the Insertion sort demonstrates how the algorithm functions:

```
Sub Insertion (V, L, H )

     J = L + 1
     DO WHILE ( J <= H )
          K = J
          DO WHILE ( K > L AND V( K ) _
               < V( K - 1 ) )
               TEMP = V( K )
               V( K ) = V( K - 1 )
               V( K - 1 ) = TEMP
               K = K - 1
          LOOP
          J = J + 1
     LOOP
End Sub
```

Sorting algorithms aren't always simple to figure out. The best way to understand a sort is to step through the logic with a small set of data. Let's use the array we used in the Bubble sort to see how the Insertion sort works (see Figure 4).

Here's the Insertion sort's step-by-step sequence:

```
point J to element 1
point K to J
If K > L, compare element K (12) _
     to element K-1 (-5) : A
don't swap because 12 > -5
```

CONTINUED FROM PAGE 42.

```
'***QUICKSORT***
Sub QuickSort (Array(), L, R)
Dim I, J, X, Y

   I = L
   J = R
   X = Array((L + R) / 2)

   While (I <= J)
   While (Array(I) < X And I < R)
     I = I + 1
   Wend
   While (X < Array(J) And J > L)
     J = J - 1
   Wend
   If (I <= J) Then
     Y = Array(I)
     Array(I) = Array(J)
     Array(J) = Y
     I = I + 1
     J = J - 1
   End If
   gIterations = gIterations + 1
   Wend

   If (L < J) Then Call _
     QuickSort(Array(), L, J)
   If (I < R) Then Call _
     QuickSort(Array(), I, R)

End Sub

'***SELECTION***
Sub Selection (Array(), ByVal _
  nOrder As Integer)
Dim Index
Dim Min
Dim NextElement
Dim TEMP

   NextElement = 0
   While (NextElement < _
     UBound(Array))
   Min = UBound(Array)
   Index = Min - 1
   While (Index >= NextElement)
     If nOrder = ASCENDING_ORDER _
       Then
       If Array(Index) < Array(Min) _
         Then
         Min = Index
       End If
       ElseIf nOrder = _
         DESCENDING_ORDER Then
       If Array(Index) >= Array(Min) _
         Then
         Min = Index
       End If
       End If
       Index = Index - 1
       gIterations = gIterations + 1
   Wend
   TEMP = Array(Min)
   Array(Min) = Array(NextElement)
   Array(NextElement) = TEMP
   NextElement = NextElement + 1
   gIterations = gIterations - 1
   Wend

End Sub

'***SHELL***
Sub ShellSort (Array(), ByVal
nOrder As Integer)
Dim Distance
Dim Size
Dim Index
Dim NextElement
Dim TEMP

   Size = UBound(Array) - _
     LBound(Array) + 1
   Distance = 1

   While (Distance <= Size)
   Distance = 2 * Distance
   Wend

   Distance = (Distance / 2) - 1

   While (Distance > 0)

   NextElement = LBound(Array) + _
     Distance

   While (NextElement <= _
     UBound(Array))
     Index = NextElement
     Do
     If Index >= (LBound(Array) + _
       Distance) Then
       If nOrder = ASCENDING_ORDER _
         Then
       If Array(Index) < _
         Array(Index - Distance) Then
         TEMP = Array(Index)
         Array(Index) = _
           Array(Index - Distance)
         Array(Index - Distance) = _
           TEMP
         Index = Index - Distance
         gIterations = gIterations _
           + 1
       Else
         Exit Do
       End If
       ElseIf nOrder = _
         DESCENDING_ORDER Then
       If Array(Index) >= _
         Array(Index - Distance) _
           Then
         TEMP = Array(Index)
         Array(Index) = _
           Array(Index - Distance)
         Array(Index - Distance) = _
           TEMP
         Index = Index - Distance
         gIterations = gIterations _
           + 1
       Else
         Exit Do
       End If
       End If
     Else
       Exit Do
     End If
     Loop
     NextElement = NextElement + 1
     gIterations = gIterations + 1
   Wend
   Distance = (Distance - 1) / 2
   gIterations = gIterations + 1
   Wend

End Sub
```

```
increment J by 1
point K to J
compare element K (7) to element _
     K-1 (12) : B
swap because 12 > 7, so decrement K _
     by 1
since K > L stay inside inner loop _
     again compare next adjacent pair
compare element K (7) to element _
     K-1 (-5) : C
don't swap because 7 > -5
increment J by 1
since J > H we have exited the array _
     and the outer loop
Array Is Sorted
```

The Insertion sort has a sibling called the Bucket sort. Like twins, these algorithms have much in common, and many subtle differences. For example, rather than performing many swaps, the Bucket sort holds an entry until it finds the correct position. Compared to the standard Insertion sort, the Bucket sort is faster, although the number of swaps performed by either sort method is similar in the end. In tests the Bucket sort actually performed more iterations than the Insertion sort, but the overall processing time was faster.

In this example, the Bucket sort holds the out-of-place value in a bucket until the correct position is located, rather than swapping entries as it iterates the array:

```
Sub Bucket (V, L, H )
     J = L + 1
     DO WHILE ( J <= H )
          BUCKET = V ( J )
          K = J
          DO WHILE ( K > L AND BUCKET _
               < V ( K - 1 ) )
               V( K ) = V( K - 1 )
               K = K - 1
          LOOP
          V ( K ) = BUCKET
          J = J + 1
     LOOP
End Sub
```

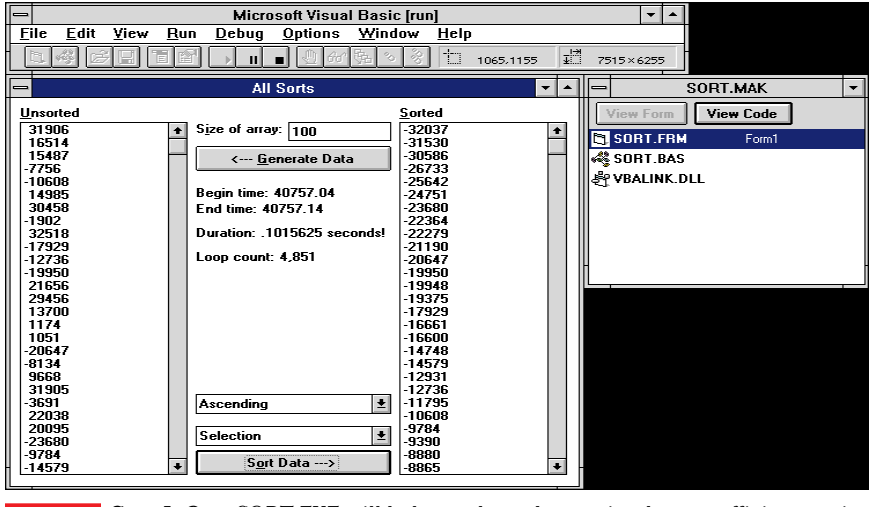

FIGURE 2 ***Sort It Out.*** *SORT.EXE will help you learn how to implement efficient sorting algorithms inside Visual Basic. It can be downloaded from the* VBPJ *Forum on CompuServe. Type GO VBPJFO and look for VBSORT.ZIP in the Magazine Library.*

HOME

Of the N ** 2 sorting algorithms, the Insertion sort is the most efficient. On the average, it will take less time than both the Selection and Bubble sorts. But the price you pay for using the Insertion sort is that it does not work well if the array is presorted in opposite order. That's where the Selection sort comes in.

The Selection sort operates on a single entry until it finds the proper location (see Figure 5). At each swap the array begins to become more sorted until the final two entries are swapped. Some algorithms suggest selecting the lowest entry to build a sorted array; others suggest the highest. The operation of the Selection sort in the example code here has not changed—only the sorting order has:

```
Sub Selection (V, L, H )

    J = L
    DO WHILE ( J < H )
        MIN = H
        K = H - 1
        DO WHILE ( K >= J )
            IF ( V ( K ) < V( MIN ) _
                ) THEN
                MIN = K
            ENDIF
            K = K - 1
        LOOP
        TEMP = V( MIN )
        V( MIN ) = V( J )
        V( J ) = TEMP
        J = J + 1
    LOOP
End Sub
```

The Selection sort is slightly faster than the Bubble sort, but not by a margin so wide that they can be considered competitors in the N ** 2 category. On large sets of data, the Selection sort has not performed well. If your applications require sorting on large arrays, the next few sort methods may be of interest.

The Shell sort, like the Selection sort, is derived from the Insertion sort and is based on the concept of diminishing increments. The Shell sort does not sort the entire array at once. First, the array is divided into smaller segments, which are then sorted separately using the Insertion sort. The sort begins with comparisons of entries located at the greatest distance possible (see Figure 6).

With each subsequent pass, the distance between compared entries becomes smaller until the last increment is one. Each pass increases the order until the array is sorted:

```
Sub Shell (V, L, H )

    D: Distance between entries
    N: Size of the array

    N = H - L + 1
    D = 1

    DO WHILE ( D <= N )
        D = 2 * D
    LOOP

    D = D/2 -1

    DO WHILE ( D > 0 )
        J = L + D
        DO WHILE ( J <= H )
            K = J
            DO WHILE ( K >= L + D _
                AND V( K ) < V( K - _
                D ) )
                TEMP = V( K )
                V( K ) = V( K - 1 )
                V( K - 1 ) = TEMP
                K = K - D
            LOOP
            J = J + 1
        LOOP
        D = ( D - 1 ) / 2
    LOOP
End Sub
```

The Shell sort is considerably faster than any of the N ** 2 sorts. In mathematical terms, the Shell sort is proportional to N ** 0.5. On large arrays, the Bubble, Insertion, Selection, and Bucket sorts do not approach the execution time of the Shell sort. Unlike the Insertion sort, whose performance would degrade on presorted arrays, the Shell sort is equally efficient in either ascending or descending order.

QUICK EXCHANGE

The Quick sort algorithm is an exchange type of sort. These sorts involve a basic idea of exchanging or switching pairs of elements, gradually moving each element closer to its final position in the array. When this technique is applied to a sort, it works equally well. Split the array into two subarrays (see Figure 7):

```
1 contains all entries < a given element
2 contains all entries > a given element
```

We continue to divide each sublist down until we reach the last element in the list. This approach is applied to Sublist 2, once Sublist 1 is completely sorted. The Quick sort lends itself very easily to using recursive techniques. The one problem in using the Quick sort is the selection of the dividing or pivot element. We want an element in the array that ideally represents the median of all elements in the list.

When you write recursive routines in Visual Basic, you need to avoid running out of stack space. Because you are not working in a C/C++ environment, you don't need direct control over the stack space, so certain limitations are imposed on Visual Basic programmers.

This Quick sort implementation always calculates the pivot point to be the center element in the target array:

```
Sub Quick ( V, Left, Right )

    I = Left
    J = Right
    Pivot = V( (Left + Right) / 2 )

    DO WHILE ( I <= J )
        DO WHILE ( V( I ) < Pivot _
            AND I < Right )
            I =  I + 1
        LOOP
        DO WHILE ( Pivot < V( J ) < _
            Pivot AND J > Left )
            J = J -1
        LOOP

        IF ( I <= J ) THEN
            Y = V( I )
            V( I ) = V( J )
            V( J ) = Y
            I = I + 1
            J = J - 1
        ENDIF
    LOOP

    IF ( Left < J ) THEN Quick( V, _
        Left, J )
    IF ( I < Right ) THEN Quick( V, _
        I, Right )

End Sub
```

Sorts on an N log N method execute extremely fast. The Shell sort is markedly more efficient than the N ** 2 methods, but the Quick sort and its members, Heap and Merge, are substantially faster. For an array of 100 elements, the average number of comparisons is roughly 200.

The Heap sort follows the structure of a binary tree. This sorting method relies on creating a heap data structure: every element at location K is greater than or equal to the elements at 2K and 2K+1. For example:

```
    or V(i/2) >= V(i) where 1 <= i _
        <= N
```

The two kinds of Heaps are:

```
Top-heavy:  V(K) >= V(2K) AND V(2K+1)
Bottom-heavy:  V(K) <= V(2K) AND _
    V(2K+1)
```

The Heap sort algorithm is dependent on three unique processes: the Heap, the Siftup, and the Siftdown routines. The Heap routine is the driving engine of the sort that calls the Siftup and Siftdown routines until the array is sorted. The Siftup and Siftdown routines each perform a specific task. The Siftup routine builds the Heap, then the Siftdown begins swapping a parent with its largest child, eventually creating the sorted Heap:

```
Sub Heap ( V, N )

    I = 2

    'This builds the HEAP
    DO WHILE ( I <= N )
        SIFTUP ( V, I )
        I = I + 1
```

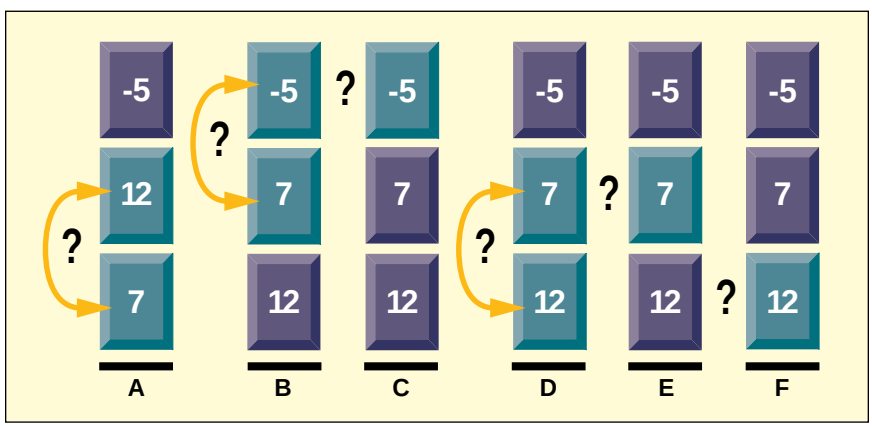

FIGURE 3 ***Bubbles Float Up.*** *The Bubble sort's objective is to rearrange the elements in the array in ascending order (low to high). It compares each member in the array with the next for each pass through the array. If time is not an issue or your array is small, the Bubble sort is the simplest to code.*

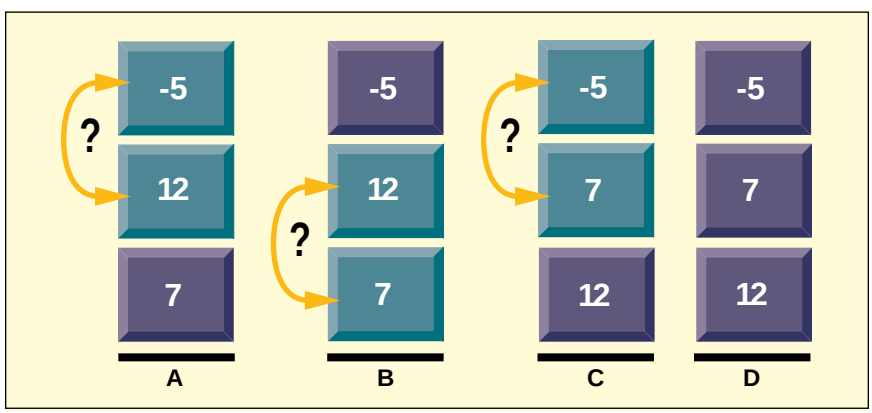

FIGURE 4 ***Better Than the Bubble.*** *Same set of data, but half the time! The Insertion sort is based on N ** 2 algorithms, but on small data sets it performs better than the Bubble sort. If you're working with large data sets, however, you need to use a more efficient algorithm instead.*

```
    LOOP
    I = N

    'by this swap the HEAP property
    'is lost; but the largest element
    'is driven to the correct
    'position. SIFTDOWN will put the
    'out of order element in correct
    'position and thus
    'create the HEAP.

DO WHILE ( I > 1 )
        TEMP = V( 1 )
        V( 1 ) = V( I )
        V( I ) = TEMP
        SIFTDOWN( V, I-1 )
        I = I - 1
    LOOP

End Sub
```

The Siftup swaps an element with its parent if it is greater than its parent and thus drives it up to its correct position:

```
Sub SIFTUP( V, M )

    I = M
    DO WHILE ( I > 1 )
        J = I / 2

        IF ( V( J ) >= V( I ) ) THEN

            BREAK
        ENDIF

        TEMP = V( I )
        V( I ) = V( J )
        V( J ) = TEMP

        I = J
    LOOP

End Sub
```

This routine is called after the highest element is swapped with the last element of the array. This routine receives the truncated array with one element out of order (from a heap). In turn, the routine drives that out-of-order element to its correct position.

The Siftdown process is logarithmic in nature because at each comparison in sifting down you eliminate one of the two substructures—that of location 2K or 2K+1:

```
Sub SIFTDOWN( V, M )

    I = 1
    J = 2 * I

    DO WHILE( J <= M )
        IF ( J < M AND V( J ) < V( _
            J + 1 ) THEN
            J = J + 1
        ENDIF

        IF V( I ) >= V( J ) THEN
            BREAK
        ENDIF

        TEMP = V( I )
        V( I ) = V( J )
        V( J ) = TEMP

        I = J
        J = 2 * I
    LOOP

End Sub
```

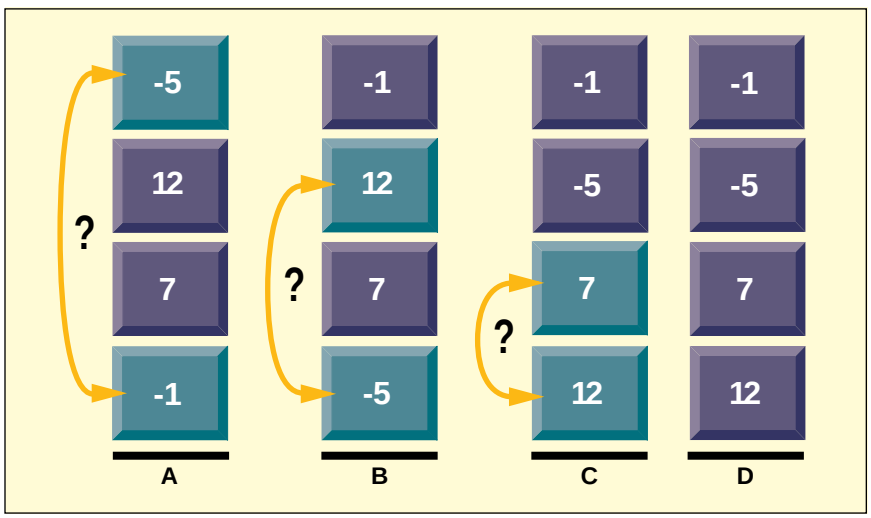

FIGURE 5 ***Pick One, Any One.*** *The Selection sort utilizes a different strategy: finding the proper location for each entry, and at each step becoming more organized. Simple in design and elegant in execution, the Selection sort is not much different from the Insertion sort.*

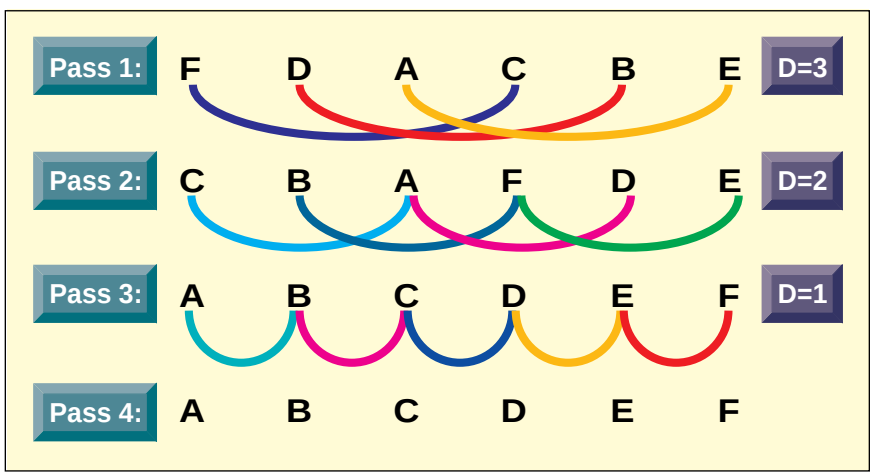

FIGURE 6 ***The Shell Game.*** *The Shell sort performs comparisons based on a distance (D). Every set of elements that are D entries apart is compared and swapped. At each pass, the distance grows smaller and the array becomes more sorted until the distance is 1, at which point the array is sorted. Shell sort will outperform the N ** 2 sorts every time on large arrays.*

The Heap sort may not be recursive like the Quick sort, but it still performs efficiently due to the Siftdown process. If you want to sort a standard large array, the Heap sort is not the best solution: use the Quick sort. If your array represents a binary tree (see Figure 8), the Heap sort is recommended. The binary tree is not a data structure used commonly by most VB programmers, but if you have been using binary trees in either VB development or another language and need a method by which you can sort the tree, give Heap sort the opportunity.

Of the sorting techniques I haven't covered, such as the two Merge sorts (recursive and nonrecursive), the nonrecursive Quick sort, the Exchange sort, and a variety of others have been derived from the core sorting algorithms presented here. Whether any of these other sorting algorithms is better than the others is a question you need to answer based on your situation.

LARGE ARRAYS AND VB DON'T MIX

I wouldn't recommend sorting large arrays with Visual Basic. C programmers

already know that these sorting algorithms will execute at lightning speeds in C compared to that in Visual Basic. The pseudocode for the sorts can easily be translated into C or assembler functions and collected into a DLL to suit your sorting needs. If you prefer to save some programming time, many third-party vendors provide libraries of DLLs that allow you to use their sorting routines instead.

My sorting needs have not be very demanding, so I prefer handwritten code in Visual Basic. The arrays most programmers sort are often small, with no more than 100 elements, so purchasing another third-party tool may not be an attractive alternative.

For huge arrays, consider using a sort written in C or you'll be waiting for minutes while the sort is executing. To see what I mean, use my SORT.EXE program (see Figure 2) to sort a small array of 4,000 elements with the Bubble sort.

Sorting can come in handy when you are dealing with a list of data that must be presented in an ordered fashion. If the luxury of a SQL database that supports ORDER BY and GROUP BY clauses or presorted data in a file isn't available, the only choice is to sort the data manually. Many of these sorting algorithms can be modified to read from a file and sort the data, or else you can read the data into an array and sort it. The size of the data should help indicate which sort to use. Avoid any of the N ** 2 arrays when sorting large arrays: the larger the array the more time it will take to sort.

For example, consider a table containing manufacturer codes for cars as well as a unique ID for each code. The table is in order by code, not by ID. It would be easy to read this table into the VB program and display the codes in a combo box. Each record is read one at a time until the end of file is reached, and the records are added to the combo box—again, the list would be ordered by code, not by ID.

If the code reads the table into a two-dimensional array and then sorts the array on the first dimension (the IDs), the array could be displayed in order by ID when the sort finished. Sorts on zip codes, social security numbers, or federal tax ID numbers, all of which are unique numbers, are often used as well.

The SORT.EXE example program demonstrates the sorts covered in this article. The nOrder parameter in the Bubble, Shell, Insertion, Bucket, and Selection sorts indicate which order the array should be sorted in: ascending or descending order. Arrays of various sizes can be built by entering a size in the text field and clicking on the Generate Data button, and selecting the sort method and sort order to sort your array.

Although variants aren't recommended by most programmers when coding, these sorting algorithms can sort more than just Integers. This approach may look poor, but I wanted to demonstrate a neat place where variants can come in handy.

For performance reasons, it would be better to build a set of unique sorts, one for each data type, but if you perform as little sorting as most programmers do, and time is not an issue, a simple generic sort is easier to maintain. The array can be composed of strings or a numeric data type, and still be sorted with the same routine.

A global counter variable is placed in each sort function to count the approximate number of iterations performed for that sort to complete. This number is not indicative of the number of logical comparisons performed for the array to be sorted. In the more elaborate sorts, the count decreases dramatically: a good sign of better performance.

If you simply need to sort a list of names and these sorting algorithms seem overdone, VB comes with a sort control. The list box has a Sorted property, which when set to True will sort its internal list in alphabetical order. Take care not to perform AddItems on the list though, which could throw off the sort.

The best way to determine which sort is right for you is to experiment with varying sizes of array and data types. No one array is the best solution; time, complexity of data type, and size of the array all play a part in selecting the right sort. My personal selection is the Bubble sort for small arrays, and the Quick sort for large arrays. Experiment with the sorting routines to see which ones you like the best. If you happen to have some of your own that aren't included in the SORT.BAS file, send them to me on CompuServe and I will include them in my SORT.BAS file for everyone else to share. ■

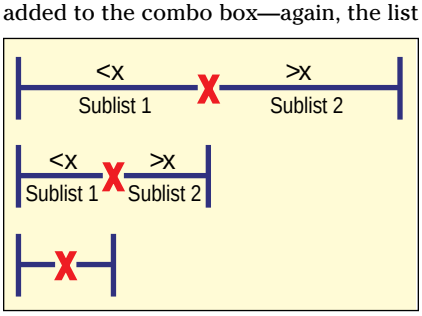

FIGURE 7 ***Quicker Than a Speeding Bullet.*** *The popular Quick sort can be written either recursively or nonrecursively. In the recursive model, the array is broken into smaller sublists, based on a Pivot point, and are then sorted. Use the Quick sort for large data sets that must be sorted quickly, but beware of stack space limitations in VB with a recursive routine!*

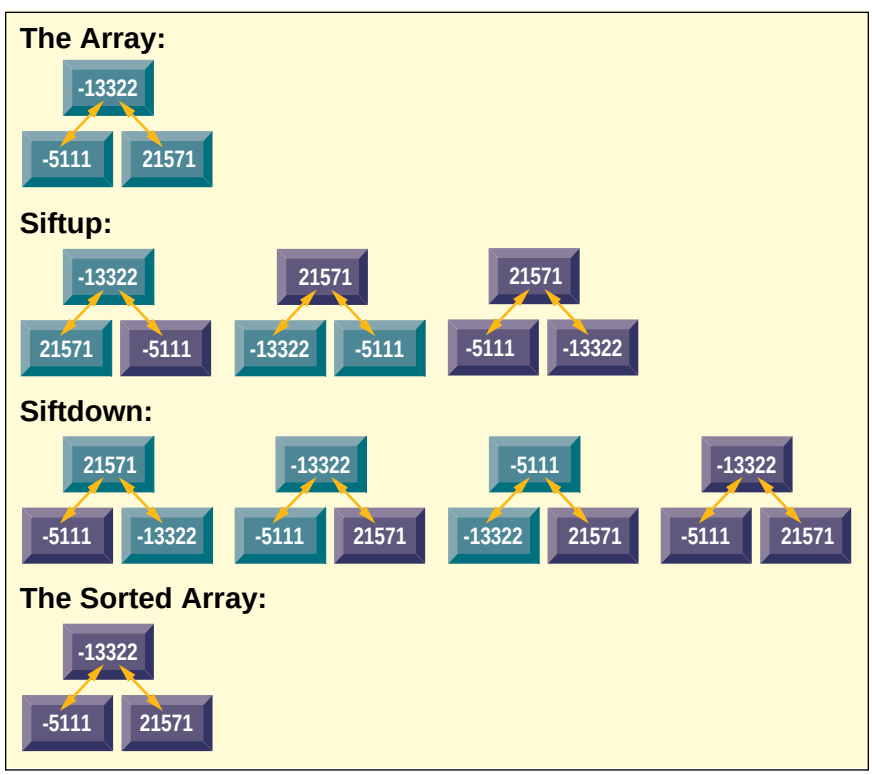

FIGURE 8 ***The Heap is On.*** *Remember all the trouble you had with binary trees when you were still in school? The Heap sort is built on the principle of balancing a binary tree. Phase I is to build the heap. Phase II is to sort the tree. The Heap sort is not recursive, but it's still fast.*