---

procedure *QuickSort*(**table** $A[l..r]$):
{Sort $A[l..r]$. The outermost call should be *QuickSort*($A[0..n-1]$)}
  if $l < r$ **then**
    $i \leftarrow l$ {i scans from the left to find elements $\geq$ the pivot}
    $j \leftarrow r + 1$ {j scans from the right to find elements $\leq$ the pivot}
    $v \leftarrow A[l]$ {v is the pivot element}
    while $i < j$ **do**
      $i \leftarrow i + 1$
      while $i \leq r$ **and** $A[i] < v$ **do** $i \leftarrow i + 1$
      $j \leftarrow j - 1$
      while $j \geq l$ **and** $A[j] > v$ **do** $j \leftarrow j - 1$
      if $i \leq r$ **then** $A[i] \leftrightarrow A[j]$
    if $i \leq r$ **then** $A[i] \leftrightarrow A[j]$ {Undo extra swap}
    $A[j] \leftrightarrow A[l]$ {Move the pivot element into its proper position}
    QuickSort($A[l..j-1]$)
    QuickSort($A[j+1..r]$)

Algorithm 11.6 Quick Sort.

---

bad luck, or because the table was in order already, that element might turn out to be the smallest table element, and then the two parts would wind up very disproportionate in size. A couple of methods for avoiding this kind of imbalance are discussed below.

In Algorithm 11.6 the partitioning around the pivot element is carried out by running two scans, one from left to right in search of an element greater than or equal to the pivot, and one from right to left in search of an element less than or equal to the pivot. When two such elements are located, they are exchanged and the scan continues. The partitioning phase stops when the two scans meet each other (Figure 11.2).

Quick Sort has time complexity $O(n^2)$ in the worst case, and as implemented in Algorithm 11.6 this worst case occurs when the table is initially in order. We could try to avoid this worst case by exchanging the first and the middle element in the table before beginning the partitioning, by inserting a new step

$$A[l] \leftrightarrow A[\lfloor (l+r)/2 \rfloor]$$

at the beginning of Algorithm 11.6. Unfortunately this merely changes the permutation that leads to the worst-case performance; it does not eliminate such permutations (Problem 17).

A better variation on Algorithm 11.6 takes the first, middle, and last elements of the table, rearranges them in order, and then uses the median of the three as the partition element. This method is illustrated in Algorithm 11.7.
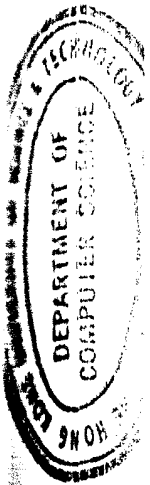
```
/* 1*/   template <class Etype>
/* 2*/   Etype &
/* 3*/   Median3( Etype A[],
/* 4*/              const unsigned int Left, const unsigned int Right )
/* 5*/   {
/* 6*/       unsigned int Center = ( Left + Right ) / 2;

/* 7*/       if( A[ Left ] > A[ Center ] )
/* 8*/           Swap( A[ Left ], A[ Center ] );
/* 9*/       if( A[ Left ] > A[ Right ] )
/*10*/           Swap( A[ Left ], A[ Right ] );
/*11*/       if( A[ Center ] > A[ Right ] )
/*12*/           Swap( A[ Center ], A[ Right ] );

/*13*/       // Invariant:  A[ Left ] <= A[ Center ] <= A[ Right ].
/*14*/       // Now hide and return pivot.

/*15*/       Swap( A[ Center ], A[ Right - 1 ] );
/*16*/       return A[ Right - 1 ];
/*17*/   }
```

Figure 7.13 Code to perform median-of-three partitioning

```
/* 1*/   template <class Etype>
/* 2*/   void
/* 3*/   Q_Sort( Etype A[],
/* 4*/              const unsigned int Left, const unsigned int Right )
/* 5*/   {
/* 6*/       if( Left + Cutoff <= Right )
/* 7*/       {
/* 8*/           Etype Pivot = Median3( A, Left, Right );
/* 9*/           unsigned int i = Left, j = Right - 1;

/*10*/           for( ; ; )
/*11*/           {
/*12*/               while( A[ ++i ] < Pivot );
/*13*/               while( A[ --j ] > Pivot );
/*14*/               if(i < j )
/*15*/                   Swap( A[ i ], A[ j ] );
/*16*/               else
/*17*/                   break;
/*18*/           }

/*19*/           Swap( A[ i ], A[ Right - 1 ] ); // Restore pivot.

/*20*/           Q_Sort( A, Left, i - 1 );
/*21*/           Q_Sort( A, i + 1, Right );
/*22*/       }
/*23*/   }
```

Figure 7.14 Main quicksort routine

(Mark Allen Weiss, 1994)