public static void mergeSort( Comparable [] a ) {
    Comparable [] tmpArray = new Comparable[ a.length ];
    mergeSort( a, tmpArray, 0, a.length - 1 );
}
private static void mergeSort( Comparable [] a, Comparable [] tmpArray, int left, int right ) {
    if( left < right ) {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }
}
private static void merge( Comparable [ ] a, Comparable [ ] tmpArray,  
    int leftPos, int rightPos, int rightEnd )  
{  
    int leftEnd = rightPos - 1;  
    int tmpPos = leftPos;  
    int numElements = rightEnd - leftPos + 1;  
    while( leftPos <= leftEnd && rightPos <= rightEnd )  
        if( a[ leftPos ].compareTo( a[ rightPos ] ) < 0 )  
            tmpArray[ tmpPos++ ] = a[ leftPos++ ];  
        else  
            tmpArray[ tmpPos++ ] = a[ rightPos++ ];  
    while( leftPos <= leftEnd )   // Copy rest of first half  
        tmpArray[ tmpPos++ ] = a[ leftPos++ ];  
    while( rightPos <= rightEnd )   // Copy rest of right half  
        tmpArray[ tmpPos++ ] = a[ rightPos++ ];  
    for( int i = 0; i < numElements; i++, rightEnd-- )  
        a[ rightEnd ] = tmpArray[ rightEnd ];  
}
Figure 8.10  Quicksort

![Quicksort Diagram]

Select pivot

Partition

Quicksort small items

Quicksort large items
Figure 8.11  Partitioning algorithm: Pivot element 6 is placed at the end.

```
8  1  4  9  0  3  5  2  7  6
```

Figure 8.12  Partitioning algorithm: i stops at large element 8; j stops at small element 2.

```
8  1  4  9  0  3  5  2  7  6
```

Figure 8.13  Partitioning algorithm: The out-of-order elements 8 and 2 are swapped.

```
2  1  4  9  0  3  5  8  7  6
```

Figure 8.14  Partitioning algorithm: i stops at large element 9; j stops at small element 5.

```
2  1  4  9  0  3  5  8  7  6
```

Figure 8.15  Partitioning algorithm: The out-of-order elements 9 and 5 are swapped.

```
2  1  4  5  0  3  9  8  7  6
```
Figure 8.16  Partitioning algorithm: i stops at large element 9; j stops at small element 3.

2 1 4 5 0 3 9 8 7 6

Figure 8.17  Partitioning algorithm: Swap pivot and element in position i.

2 1 4 5 0 3 6 8 7 9

Figure 8.18  Original array

8 1 4 9 6 3 5 2 7 0

Figure 8.19  Result of sorting three elements (first, middle, and last)

0 1 4 9 6 3 5 2 7 8

Figure 8.20  Result of swapping the pivot with the next-to-last element

0 1 4 9 7 3 5 2 6 8
Quicksort

public static void quicksort( Comparable [ ] a ) { quicksort( a, 0, a.length - 1 ); }
private static void quicksort( Comparable [ ] a, int low, int high )
{
    if( low + CUTOFF > high ) insertionSort( a, low, high );
    else { // Sort low, middle, high
        int middle = ( low + high ) / 2;
        if( a[ middle ].compareTo( a[ low ] ) < 0 ) swapReferences( a, low, middle );
        if( a[ high ].compareTo( a[ low ] ) < 0 ) swapReferences( a, low, high );
        if( a[ high ].compareTo( a[ middle ] ) < 0 ) swapReferences( a, middle, high );
        swapReferences( a, middle, high - 1 ); // Place pivot at position high - 1
        Comparable pivot = a[ high - 1 ];
        int i, j; // Begin partitioning
        for( i = low, j = high - 1; ; ) {
            while( a[ ++i ].compareTo( pivot ) < 0 ) /* Do nothing */ ;
            while( pivot.compareTo( a[ --j ] ) < 0 ) /* Do nothing */ ;
            if( i >= j ) break;
            swapReferences( a, i, j );
        }
        swapReferences( a, i, high - 1 );
        quicksort( a, low, i - 1 ); // Sort small elements
        quicksort( a, i + 1, high ); // Sort large elements
    }
}
Figure 20.4
Linear probing
hash table after each insertion

hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash ( 9, 10 ) = 9
Figure 20.5
Illustration of primary clustering in linear probing (b) versus no clustering (a) and the less significant secondary clustering in quadratic probing (c). Long lines represent occupied cells, and the load factor is 0.7.
Figure 20.6
A quadratic probing hash table after each insertion (note that the table size was poorly chosen because it is not a prime number).

<table>
<thead>
<tr>
<th></th>
<th>After insert 89</th>
<th>After insert 18</th>
<th>After insert 49</th>
<th>After insert 58</th>
<th>After insert 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>89</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
</tr>
</tbody>
</table>