

```

/**
 * A class that contains several sorting routines,
 * implemented as static methods.
 * Arrays are rearranged with smallest item first,
 * using compares.
 * @author Mark Allen Weiss
 */
public final class Sort
{
    /**
     * Simple insertion sort.
     * @param a an array of Comparable items.
     */
    public static void insertionSort( Comparable [ ] a )
    {
        for( int p = 1; p < a.length; p++ )
        {
            Comparable tmp = a[ p ];
            int j = p;

            for( ; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
                a[ j ] = a[ j - 1 ];
            a[ j ] = tmp;
        }
    }

    /**
     * Shellsort, using a sequence suggested by Gonnet.
     * @param a an array of Comparable items.
     */
    public static void shellsort( Comparable [ ] a )
    {
        for( int gap = a.length / 2; gap > 0;
            gap = gap == 2 ? 1 : (int) ( gap / 2.2 ) )
            for( int i = gap; i < a.length; i++ )
            {
                Comparable tmp = a[ i ];
                int j = i;

                for( ; j >= gap && tmp.compareTo( a[ j - gap ] ) < 0; j -= gap )
                    a[ j ] = a[ j - gap ];
                a[ j ] = tmp;
            }
    }

    /**
     * Standard heapsort.
     * @param a an array of Comparable items.
     */
    public static void heapsort( Comparable [ ] a )
    {
        for( int i = a.length / 2; i >= 0; i-- ) /* buildHeap */
            percDown( a, i, a.length );
        for( int i = a.length - 1; i > 0; i-- )
        {
            swapReferences( a, 0, i ); /* deleteMax */
            percDown( a, 0, i );
        }
    }
}
/**

```

```

* Internal method for heapsort.
* @param i the index of an item in the heap.
* @return the index of the left child.
*/
private static int leftChild( int i )
{
    return 2 * i + 1;
}

/**
* Internal method for heapsort that is used in
* deleteMax and buildHeap.
* @param a an array of Comparable items.
* @index i the position from which to percolate down.
* @int n the logical size of the binary heap.
*/
private static void percDown( Comparable [ ] a, int i, int n )
{
    int child;
    Comparable tmp;

    for( tmp = a[ i ]; leftChild( i ) < n; i = child )
    {
        child = leftChild( i );
        if( child != n - 1 && a[ child ].compareTo( a[ child + 1 ] ) < 0 )
            child++;
        if( tmp.compareTo( a[ child ] ) < 0 )
            a[ i ] = a[ child ];
        else
            break;
    }
    a[ i ] = tmp;
}

/**
* Mergesort algorithm.
* @param a an array of Comparable items.
*/
public static void mergeSort( Comparable [ ] a )
{
    Comparable [ ] tmpArray = new Comparable[ a.length ];
    mergeSort( a, tmpArray, 0, a.length - 1 );
}

/**
* Internal method that makes recursive calls.
* @param a an array of Comparable items.
* @param tmpArray an array to place the merged result.
* @param left the left-most index of the subarray.
* @param right the right-most index of the subarray.
*/
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 );
    }
}

```

```

}

/**
 * Internal method that merges two sorted halves of a subarray.
 * @param a an array of Comparable items.
 * @param tmpArray an array to place the merged result.
 * @param leftPos the left-most index of the subarray.
 * @param rightPos the index of the start of the second half.
 * @param rightEnd the right-most index of the subarray.
 */
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;

    // Main loop
    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++ ];

    // Copy tmpArray back
    for( int i = 0; i < numElements; i++, rightEnd-- )
        a[ rightEnd ] = tmpArray[ rightEnd ];
}

/**
 * Quicksort algorithm.
 * @param a an array of Comparable items.
 */
public static void quicksort( Comparable [ ] a )
{
    quicksort( a, 0, a.length - 1 );
}

private static final int CUTOFF = 10;

/**
 * Method to swap two elements in an array.
 * @param a an array of objects.
 * @param index1 the index of the first object.
 * @param index2 the index of the second object.
 */
public static final void swapReferences( Object [ ] a, int index1, int index2 )
{
    Object tmp = a[ index1 ];
    a[ index1 ] = a[ index2 ];
    a[ index2 ] = tmp;
}

/**
 * Internal quicksort method that makes recursive calls.

```

```

* Uses median-of-three partitioning and a cutoff of 10.
* @param a an array of Comparable items.
* @param low the left-most index of the subarray.
* @param high the right-most index of the subarray.
*/
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 );

        // Place pivot at position high - 1
        swapReferences( a, middle, high - 1 );
        Comparable pivot = a[ high - 1 ];

        // Begin partitioning
        int i, j;
        for( i = low, j = high - 1; ; )
        {
            while( a[ ++i ].compareTo( pivot ) < 0 )
                ;
            while( pivot.compareTo( a[ --j ] ) < 0 )
                ;
            if( i >= j )
                break;
            swapReferences( a, i, j );
        }

        // Restore pivot
        swapReferences( a, i, high - 1 );

        quicksort( a, low, i - 1 );    // Sort small elements
        quicksort( a, i + 1, high );  // Sort large elements
    }
}

/**
* Internal insertion sort routine for subarrays
* that is used by quicksort.
* @param a an array of Comparable items.
* @param low the left-most index of the subarray.
* @param n the number of items to sort.
*/
private static void insertionSort( Comparable [ ] a, int low, int high )
{
    for( int p = low + 1; p <= high; p++ )
    {
        Comparable tmp = a[ p ];
        int j;

        for( j = p; j > low && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
            a[ j ] = a[ j - 1 ];
    }
}

```

```

        a[ j ] = tmp;
    }
}

/**
 * Quick selection algorithm.
 * Places the kth smallest item in a[k-1].
 * @param a an array of Comparable items.
 * @param k the desired rank (1 is minimum) in the entire array.
 */
public static void quickSelect( Comparable [ ] a, int k )
{
    quickSelect( a, 0, a.length - 1, k );
}

/**
 * Internal selection method that makes recursive calls.
 * Uses median-of-three partitioning and a cutoff of 10.
 * Places the kth smallest item in a[k-1].
 * @param a an array of Comparable items.
 * @param low the left-most index of the subarray.
 * @param high the right-most index of the subarray.
 * @param k the desired rank (1 is minimum) in the entire array.
 */
private static void quickSelect( Comparable [ ] a, int low, int high, int k )
{
    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 );

        // Place pivot at position high - 1
        swapReferences( a, middle, high - 1 );
        Comparable pivot = a[ high - 1 ];

        // Begin partitioning
        int i, j;
        for( i = low, j = high - 1; ; )
        {
            while( a[ ++i ].compareTo( pivot ) < 0 )
                ;
            while( pivot.compareTo( a[ --j ] ) < 0 )
                ;
            if( i >= j )
                break;
            swapReferences( a, i, j );
        }

        // Restore pivot
        swapReferences( a, i, high - 1 );

        // Recurse; only this part changes
        if( k <= i )

```

```
        quickSelect( a, low, i - 1, k );  
    else if( k > i + 1 )  
        quickSelect( a, i + 1, high, k );  
    }  
}  
}
```