The merge sort algorithm is recursive. It is therefore most naturally implemented as a [recursive method](https://www.eimacs.com/eimacs/mainpage?epid=E205325308&cid=162149#RecMethod). The basic idea is to split the initial array into two parts of approximately the same length. Each of these parts is then sorted (using the same merge sort technique), and the two sorted halves are then merged into a single sorted array. (For this reason, the algorithm is said to employ a "divide and conquer" technique. We will meet another divide-and-conquer sorting algorithm — [quicksort](https://www.eimacs.com/eimacs/mainpage?epid=E2247656864&cid=162149) — in the Advanced Topics section of this course.)

For this to succeed, we will need an algorithm for merging two sorted arrays into a single sorted array. In the following exercise, you will define a static method that implements this merge procedure.

**Exercise 197**

Complete the following definition of the static method merge, which merges two individually sorted arrays into a single sorted array. For example, when it is applied to the sorted arrays

[ 1, 3, 3, 5 ] and [ 2, 3, 4, 7, 9, 11 ]

and an empty result array of a suitable length, the merge method should cause the result array to become

[ 1, 2, 3, 3, 3, 4, 5, 7, 9, 11 ].

(See the main method in the code below.)

This is a challenging programming task and many different solutions are possible. It is worth spending some time to try and develop your own definition. However, if after much effort you meet with no success, click [here](javascript:secWindow('mainpage?epid=E1951961328&cid=162149&noslink=E1924903320&s=2','HintPopUp',640,480,50,50,'menubar,scrollbars,resizable')) for a hint.

public class MainClass   
{   
  // precondition: a and b are sorted in ascending order;    
  // they are NOT necessarily of the same length.   
  // postcondition: result contains all the elements of a and b,   
  // sorted into ascending order.

  public static void merge( int[] a, int[] b, int[] result )

   {

     int i,j = 0;

     for ( i = 0 ; i < a.length ; i++ )

     {

       while ( j < b.length )

       {

         if ( a[ i ] < b[ j ] )

         {

           break;

         }

         else

         {

           result[ i + j ] = b[ j ];

           j++;

         }

       }

       result[ i + j ] = a[ i ];

     }

     while ( j < b.length )

     {

       result[ i + j ] = b[ j ];

       j++;

     }

   }

public static void main( String[] args )

{

  int[] a = { 1, 3, 3, 5 };

  int[] b = { 2, 3, 4, 7, 9, 11 };

  int[] result = new int[ a.length + b.length ];

  merge( a, b, result );

  for ( int i : result )

    System.out.print( i + " " );

}

}

1 2 3 3 3 4 5 7 9 11

Our eventual goal is to implement the merge sort algorithm as a recursive method. Before reaching that level, however, let us see how the principal stages of the algorithm fit together by considering a non-recursive variation. The stages we want to focus upon are these:

1. Split the initial array into two sub-arrays of approximately the same length.
2. Sort each of the two sub-arrays.
3. Merge the two sorted sub-arrays into a single sorted array.

Eventually, the sorting in Stage 2 will be performed by recursive calls to the merge sort method. But for now let us sort the sub-arrays using a selection sort.

#### Exercise 198

Complete the following program so that it implements the merge sort variation described above. (You could use the selection sort method you produced in answer to [Exercise 184](https://www.eimacs.com/eimacs/mainpage?epid=E204488433&cid=162149#Exe171) and the merge method you wrote in answer to [Exercise 197](https://www.eimacs.com/eimacs/mainpage?epid=E2094884660&cid=162149#Exe184).)

public class MainClass   
{   
  public static void mergeSortVariation( int[] array )   
  {   
    // split the array into two   
    int mid = array.length / 2;        
    int[] a = new int[ mid ];   
    for ( int i = 0; i < mid ; i++ )   
      a[ i ] = array[ i ];   
  
    int[] b = new int[ array.length - mid ];   
    for ( int i = mid ; i < array.length ; i++ )   
      b[ i - mid ] = array[ i ];   
  
    // sort the subarrays   
    selectionSort( a );   
    selectionSort( b );   
  
    // merge the sorted subarrays back into the array   
    merge( a, b, array );   
  }   
 public static void selectionSort( int[] a )

 {

   int i, k, posmax;

   for ( i = a.length - 1 ; i > 0 ; i-- )

   {

     posmax = 0;

     for ( k = 1 ; k <= i ; k++ )

     {

       if ( a[ k ] > a[ posmax ] )

         posmax = k;

     }

     int temp = a[ i ];

     a[ i ] = a[ posmax ];

     a[ posmax ] = temp;

   }

 }

 public static void merge( int[] a, int[] b, int[] result )

 {

   int i,j = 0;

   boolean bLoop = true;

   for ( i = 0 ; i < a.length ; i++ )

   {

     while ( bLoop && j < b.length )

     {

       if ( a[ i ] < b[ j ] )

       {

         bLoop = false;

       }

       else

       {

         result[ i + j ] = b[ j ];

         j++;

       }

     }

     result[ i + j ] = a[ i ];

     bLoop = true;

   }

   while ( j < b.length )

   {

     result[ i + j ] = b[ j ];

     j++;

   }

 }

public static void main( String[] args )

{

  int[] array = { 1, 4, 3, 9, 6, 2, 3, 7, 10 };

  mergeSortVariation( array );

  // display the result

  for ( int i : array )

    System.out.print( i + " " );

}

}

1 2 3 3 4 6 7 9 10

In [Exercise 198](https://www.eimacs.com/eimacs/mainpage?epid=E1971628230&cid=162149#Exe185), you completed the definition of a variation of the merge sort in which our attention was focused on the three main stages of the sorting algorithm. In that variation, the two halves of the initial array were sorted using a selection sort before being merged back together. The method used to sort the half-arrays was of very little concern to us; all that mattered was that they should be sorted prior to being merged together. In fact, since the effect of the mergeSortVariation method is to sort the array it is given as its argument, we could use it to sort the half-arrays rather than the selectionSort method, which, after all, achieves the same sorted result by other means. If we were to do this, then the mergeSortVariation method would be calling itself, and the definition would have become recursive. But that was our goal in the first place. So, in recognition of this fact, we will simplify the method name to mergeSort.

There is one small — but very significant — technicality that has yet to be taken care of in the recursive definition that results from replacing the two calls to selectionSort by calls to the mergeSort method itself. The method doesn't know when to stop! The code tells it to split and sort, split and sort, and to keep on doing that. Of course, at each split, each of the resulting sub-arrays is approximately half as long as the array that is split. Eventually, this produces a sub-array of length 1, containing only a single element. Such an array is of course already sorted without any further action being required. So to cause the mergeSort method to stop we have only to tell it to return immediately, doing no splitting and sorting, if the argument is an array of length 1.

We present the resulting recursive merge sort method below. There are two things to notice about this code:

* We have included statements whose purpose is to display the array at each call to mergeSort. This should help you trace the progress of the sort.
* We present a version of the merge method that is different from the one we provided in our solution to [Exercise 197](https://www.eimacs.com/eimacs/mainpage?epid=E206162822&cid=162149#Exe184) and, very likely, different from your solution to that exercise. It features a while-loop whose condition is the boolean true. Such a loop will of course cycle around endlessly unless there is some way to break out of it that is independent of the loop condition. In our case, this is provided by an empty return statement that is reached when the value of index j becomes equal to the length of array b.

Click the **Run** button and study the output carefully. Try to figure out from the output the order in which the sub-arrays are sorted and merged. This is complicated, so don't worry if you get lost — we investigate the matter more thoroughly on the next page.

(The displayIntArray method used in the main method is provided in hidden code. To view its definition, click the Show program details link.)

public class MainClass   
{   
  public static void displayIntArray( int[] array )   
  {   
    for ( int i : array )   
      System.out.print( i + " " );   
  
    System.out.println();   
  }   
    
  public static void mergeSort( int[] array )   
  {   
    System.out.print( "Sorting " );   
    displayIntArray( array );     
  
    // stop when array has only 1 element   
    if ( array.length == 1 )   
      return;   
  
    // split array into two   
    int mid = array.length / 2;        
    int[] a = new int[ mid ];   
    for ( int i = 0; i < mid ; i++ )   
      a[ i ] = array[ i ];   
  
    int[] b = new int[ array.length - mid ];   
    for ( int i = mid ; i < array.length ; i++ )   
      b[ i - mid ] = array[ i ];   
  
    // sort the sub-arrays   
    mergeSort( a );   
    mergeSort( b );   
  
    // merge the sorted sub-arrays back into the array   
    merge( a, b, array );   
  
    System.out.print( " Result " );   
    displayIntArray( array );     
  }   
    
  public static void merge( int[] a, int[] b, int[] result )  
  {  
    int i = 0, j = 0;  
    
    while ( true )  
    {  
      if ( i == a.length )  
      {  
        if ( j == b.length )  
          return;  
    
        result[ i + j ] = b[ j ];   
        j++;  
      }  
      else if ( j == b.length )  
      {  
        result[ i + j ] = a[ i ];  
        i++;  
      }  
      else if ( a[ i ] < b[ j ] )  
      {  
        result[ i + j ] = a[ i ];  
        i++;  
      }  
      else  
      {  
        result[ i + j ] = b[ j ];  
        j++;  
      }  
    }  
  }  
    
  public static void main( String[] args )  
  {  
    int[] array = { 1, 4, 3, 9, 6, 2, 3, 10, 7, 2 };  
    mergeSort( array );  
    
    displayIntArray( array );  
  }    
}

Sorting 1 4 3 9 6 2 3 10 7 2    
Sorting 1 4 3 9 6    
Sorting 1 4    
Sorting 1    
Sorting 4    
 Result 1 4    
Sorting 3 9 6    
Sorting 3    
Sorting 9 6    
Sorting 9    
Sorting 6    
 Result 6 9    
 Result 3 6 9    
 Result 1 3 4 6 9    
Sorting 2 3 10 7 2    
Sorting 2 3    
Sorting 2    
Sorting 3    
 Result 2 3    
Sorting 10 7 2    
Sorting 10    
Sorting 7 2    
Sorting 7    
Sorting 2    
 Result 2 7    
 Result 2 7 10    
 Result 2 2 3 7 10    
 Result 1 2 2 3 3 4 6 7 9 10    
1 2 2 3 3 4 6 7 9 10

To give you a deeper understanding of how a merge sort progresses, we work our way step by step through the process of applying a merge sort to the following array. (At each stage, the sub-array being sorted has a red border. As a sub-array becomes sorted, its border turns green.)

|  |
| --- |
| 1, 8, 5, 3, 4, 9, 7, 2, 6 |

The array contains nine elements, so we split it into a 4-element sub-array and a 5-element sub-array, like this:

|  |
| --- |
| 1, 8, 5, 3  4, 9, 7, 2, 6 |

We now sort the sub-array highlighted in red. It has four elements, so we split it into two sub-arrays of two elements each:

|  |
| --- |
| 1, 8  5, 3  4, 9, 7, 2, 6 |

We continue by sorting the sub-array highlighted in red. This has just two elements, so we split it into two sub-arrays each having one element:

|  |
| --- |
| 1  8  5, 3  4, 9, 7, 2, 6 |

The red-highlighted sub-array has only one element, so it is sorted and we leave it unchanged. We then move on to the second sub-array:

|  |
| --- |
| 1  8  5, 3  4, 9, 7, 2, 6 |

This too contains only one element, so it is sorted and we leave it unchanged:

|  |
| --- |
| 1  8  5, 3  4, 9, 7, 2, 6 |

We now merge these two sorted sub-arrays:

|  |
| --- |
| 1, 8  5, 3  4, 9, 7, 2, 6 |

We continue by sorting the red-highlighted sub-array. This has just two elements, so we split it into two sub-arrays each having one element:

|  |
| --- |
| 1, 8  5  3  4, 9, 7, 2, 6 |

The red-highlighted sub-array has just one element, so it is sorted and we leave it unchanged. We then move on to the second sub-array:

|  |
| --- |
| 1, 8  5  3  4, 9, 7, 2, 6 |

This too contains just one element, so it is sorted and we leave it unchanged:

|  |
| --- |
| 1, 8  5  3  4, 9, 7, 2, 6 |

We now merge the two most recent sub-arrays:

|  |
| --- |
| 1, 8  3, 5  4, 9, 7, 2, 6 |

This results in two neighboring, sorted sub-arrays, which we now merge:

|  |
| --- |
| 1, 3, 5, 8  4, 9, 7, 2, 6 |

This complete the sorting of the left-hand sub-array of the original array. A similar process now takes place with the right-hand sub-array. Here are the steps:

|  |
| --- |
| 1, 3, 5, 8  4, 9  7, 2, 6 |
| 1, 3, 5, 8  4  9  7, 2, 6 |
| 1, 3, 5, 8  4  9  7, 2, 6 |
| 1, 3, 5, 8  4  9  7, 2, 6 |
| 1, 3, 5, 8  4, 9  7, 2, 6 |
| 1, 3, 5, 8  4, 9  7  2, 6 |
| 1, 3, 5, 8  4, 9  7  2, 6 |
| 1, 3, 5, 8  4, 9  7  2  6 |
| 1, 3, 5, 8  4, 9  7  2  6 |
| 1, 3, 5, 8  4, 9  7  2  6 |
| 1, 3, 5, 8  4, 9  7  2, 6 |
| 1, 3, 5, 8  4, 9  2, 6, 7 |
| 1, 3, 5, 8  2, 4, 6, 7 9 |
| 1, 2, 3, 4, 5, 6, 7, 8, 9 |

If you would like to view a Java applet that presents an animation of a merge sort, click [here](javascript:secWindow('http://www.cosc.canterbury.ac.nz/people/mukundan/dsal/MSort.html','AnimPopUp',715,610,0,0,'menubar,scrollbars,resizable')). (Be patient while the Java applet loads.)

public class MainClass   
{   
  public static void mergeSort( int[] a, int low, int high )   
  {   
    if ( low >= high )   
      return;   
  
    int mid = ( low + high ) / 2;   
  
    mergeSort( a, low, mid );    
    mergeSort( a, mid + 1, high );    
  
    merge( a, low, mid, high );   
  }

public static void merge( int[] a, int low, int mid, int high )

 {

   int[] temp = new int[ high - low + 1 ];

   int i = low, j = mid + 1, n = 0;

   while ( i <= mid || j <= high )

   {

     if ( i > mid )

     {

       temp[ n ] = a[ j ];

       j++;

     }

     else if ( j > high )

     {

       temp[ n ] = a[ i ];

       i++;

     }

     else if ( a[ i ] < a[ j ] )

     {

       temp[ n ] = a[ i ];

       i++;

     }

     else

     {

       temp[ n ] = a[ j ];

       j++;

     }

     n++;

   }

   for ( int k = low ; k <= high ; k++ )

     a[ k ] = temp[ k - low ];

 }

  public static void main( String[] args )   
  {   
    Random r = new Random();   
    int[] array = new int[ 25 ];   
  
    System.out.print( "Before: " );   
    for ( int i = 0 ; i < array.length ; i++ )   
    {   
      array[ i ] = r.nextInt( 125 ) - 25;   
      System.out.print( array[ i ] + " " );   
    }   
  
    mergeSort( array, 0, array.length - 1 );   
  
    System.out.print( "\n After: " );   
    for ( int t : array )   
      System.out.print( t + " " );   
  }   
}

Before: 77 -19 94 17 -25 35 10 7 72 57 95 92 -11 89 -3 39 85 80 43 93 67 -13 -24 98 76    
 After: -25 -24 -19 -13 -11 -3 7 10 17 35 39 43 57 67 72 76 77 80 85 89 92 93 94 95 98

It may be shown that applying a merge sort to an array of length *n* requires, on average, *n* log2 *n* comparisons. We say that a merge sort executes "in *n*-log-*n* time". For large arrays this represents a very significant improvement in efficiency over selection sorts and insertion sorts, as the following table shows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Average time** [if each comparison takes 1 millisecond] | | | |
| **Array size** | **Selection sort** | **Insertion sort** | **Merge sort** |
| 16 | 0.12 s | 0.068 s | 0.064 s |
| 1024 | 8 min 44 s | 4 min 22 s | 10.24 s |
| 250,000 | 361 days 16 h | 180 days 20 h | 1 h 15 min |
| 1,000,000 | 15 years 312 days | 7 years 338 days | 5 h 32 min |

Looked at from a different point of view, a merge sort running on a small personal computer will easily beat a supercomputer running a selection sort on a large array of, say, 1,000,000 integers.

For more information on the subject of merge sorts, visit the National Institute of Standards and Technology Dictionary of Algorithms and Data Structures [Merge sort](http://xlinux.nist.gov/dads/HTML/mergesort.html) page.

896