A common task in computing is that of sorting the elements of an array into order — in other words, arranging the elements in ascending or descending order according to some property of the elements.

There are a great many sorting algorithms. Some are more efficient than others, some have more complex code than others, and — as was the case for searching algorithms — the choice of a suitable algorithm will often depend upon how many elements are to be sorted and what their data type is. On the next few pages, we consider three algorithms for sorting: the *selection sort*, the *insertion sort*, and the *merge sort*. (In a later section that deals with more advanced topics for students who enjoy a challenge, we also meet three more sorting algorithms: the [*quicksort*](https://www.eimacs.com/eimacs/mainpage?epid=E2222769504&cid=162149), the [*tree sort*](https://www.eimacs.com/eimacs/mainpage?epid=E2163748540&cid=162149#TreeSort), and the [*heapsort*](https://www.eimacs.com/eimacs/mainpage?epid=E2395215520&cid=162149).)

One of the simplest of these sorting algorithms is the *selection sort*. Suppose we are sorting the following five-element array of ints into ascending order:

Since we are sorting the five integers into ascending order, once the sort is complete the smallest of them (11) will be the first array element and the largest (15) will be the last.

The algorithm works as follows: First, we iterate through the array to find the position of the largest element. In this case, it's the element at index 2, namely 15. This element needs to be placed at the end of the array. But there's already an element (13) there, so to avoid overwriting 13 we swap it with the newly-found largest element. This puts 15 at the end of the array (where it needs to be) and 13 in place of 15 at index 2:

At this point, the largest element is already in its final resting place. We then repeat the process, except that we ignore the last element. That is, we pretend that the array now only contains four elements — as though its last element is at index 3 — and repeat our steps. So we look among the first four elements to find the largest; it's 14, at index 1. We then make this element the "last" element by swapping it with the element currently at index 3, namely 11. 14 is now in its final resting place:

We then repeat the process again except that, since the final two array elements have already been sorted, we now only consider the first three elements. The largest of these is 13 at index 2. It is already in the right spot, so there is no need to move it. However, for consistency's sake, we can say that we "swap" it with itself. We then consider the first two elements only. The larger of these is 12 at index 0. So we swap it with 11 at index 1, and the array is sorted:

To help us as we encode this algorithm, let us work with a particular array, defined as follows:

  int[] array = { 12, 14, 15, 11, 13 };

Since there are five elements in this array and since an element is placed into its correct position at each iteration, we will iterate five times using a for-loop with i as the counter. To make it easier to ignore the last element at each iteration, we will arrange things so that i counts down from 4 to 0, like this:

  int i;   
  for ( i = 4 ; i >= 0 ; i-- )   
  {   
    // ...   
  }

As you will see, this means that, during each iteration, array[ i ] is always the last of the array elements that have not yet been sorted.

We now wish to find the largest element of the array. We do this by iterating through the array using a for-loop with k as the counter, and using a variable posmax to track the index of the largest element we have found so far, like this:

  int k, posmax = 0;   
  for ( k = 0 ; k < 5 ; k++ )   
  {   
    if ( array[ k ] > array[ posmax ] )   
      posmax = k;   
  }

Once this code has executed, posmax will be the index of the largest element. In our sample array, 15 is the largest element, so posmax will be 2. We swap this element with the element currently at the end of the array, that is, the element at index i, namely 13:

  int temp = array[ i ]; // the last element   
  array[ i ] = array[ posmax ];   
  array[ posmax ] = temp;

We now repeat the process. The code so far is shown below, but in fact it does not work correctly. Use the single stepper to help you figure out what is going wrong.

int[] array = { 12, 14, 15, 11, 13 };

int i, k, posmax;

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

 {

   posmax = 0;

   for ( k = 0 ; k <= i ; k++ )

   {

     if ( array[ k ] > array[ posmax ] )

       posmax = k;

   }

   int temp = array[ i ];

   array[ i ] = array[ posmax ];

   array[ posmax ] = temp;

 }

The code in the single stepper below sorts an array of three elements. Single-step through the code, paying especially close attention during the final iteration, when the value of i is 0. In particular, note

* the state of the array during that iteration, and
* the values of k and posmax when the final array[ k ] > array[ posmax ] comparison is made.

**Single Stepper**

  int[] array = { 3, 1, 2 };  
  int i;  
  int k;  
  int posmax;  
  int temp;  
  
  for ( i = array.length - 1 ; i >= 0 ; i-- )  
  {  
    posmax = 0;  
  
    for ( k = 0 ; k <= i ; k++ )  
    {  
      if ( array[ k ] > array[ posmax ] )  
        posmax = k;  
    }  
  
    temp = array[ i ];  
    array[ i ] = array[ posmax ];  
    array[ posmax ] = temp;  
  }

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Watch**   |  |  | | --- | --- | | array | {1, 2, 3} | | i | -1 | | k | 1 | | posmax | 0 | | temp | 1 | | **Output** |

**Notes:** The program has finished.

You probably noticed that, when the outer for-loop begins its last iteration, the array is already sorted. This is always the case, for when i is 1 the algorithm has sorted all but the first two elements of the array. It then finds the larger of those two elements and places it at index 1, leaving an element at index 0 that is not greater than any other element. But this means that the element at index 0 is located exactly where it should be, and the array is already sorted. There is therefore no need to execute the iteration when i is 0.

In case you need further persuasion, observe that, since i is 0 during the final iteration, the inner for-loop only iterates once and that in this iteration k is also 0. So the final comparison in the inner for-loop is always array[ 0 ] > array[ 0 ], which is always false. As a result, the body of the if-statement in the inner for-loop is never executed during the final interation, thus making that iteration a waste of time.

Having had your attention drawn to the inner for-loop, you should now realize that it too performs a wasted iteration. Each time it is executed, the variable posmax has just been initialized to 0. During the first iteration of the for-loop, the variable k has the value 0. So the first comparison is always array[ 0 ] > array[ 0 ], which — once again — is always false.

We may therefore improve the efficiency of the selection sort algorithm by making the last iteration of the outer for-loop be the one when i is 1 and by making the first iteration of the inner for-loop be the one when k is 1.

public static void selectionSort( int[] array )  
{  
  int i, k, posmax;  
  
  for ( i = array.length - 1 ; i > 0 ; i-- )  
  {  
    posmax = 0;  
  
    for ( k = 1 ; k <= i ; k++ )  
    {  
      if ( array[ k ] > array[ posmax ] )  
        posmax = k;  
    }  
  
    int temp = array[ i ];  
    array[ i ] = array[ posmax ];  
    array[ posmax ] = temp;  
  }  
}

Modify the definition of selectionSort below so that the array is sorted into descending order. Test your code thoroughly using various arrays of different lengths.

public static void selectionSort( int[] array )  
{  
  int i, k, posmax;  
  
  for ( i = array.length - 1 ; i > 0 ; i-- )  
  {  
    posmax = 0;  
  
    for ( k = 1 ; k <= i ; k++ )  
    {  
      if ( array[ k ] < array[ posmax ] )  
        posmax = k;  
    }  
  
    int temp = array[ i ];  
    array[ i ] = array[ posmax ];  
    array[ posmax ] = temp;  
  }  
}

Although it is not necessary in order to make the code work as intended, it would make the code easier for a human being to read and understand if each occurrence of the variable posmax in this code were replaced by one, such as posmin, that is more descriptive of its role in a descending order sort.

The selectionSort method below is intended to sort a String array into ascending order using the lexicographic order. Complete the definition and test your code.

The definition is virtually identical to the one for sorting arrays of ints, the only differences being that the comparison in the inner for-loop must use the compareTo method instead of > and the variable temp must be declared as a String instead of an int.

public static void selectionSort( String[] array )  
{  
  int i, k, posmax;  
  
  for ( i = array.length - 1 ; i > 0 ; i-- )  
  {  
    posmax = 0;  
  
    for ( k = 1 ; k <= i ; k++ )  
    {  
      if ( array[ k ].compareTo( array[ posmax ] ) > 0 )  
        posmax = k;  
    }  
  
    String temp = array[ i ];  
    array[ i ] = array[ posmax ];  
    array[ posmax ] = temp;  
  }  
}

You may have noticed that if the array has *n* elements then the first iteration involves *n* – 1 comparisons, the second involves *n* – 2 comparisons, the third involves *n* – 3 comparisons, and so on until the last iteration which involves just 1 comparison. So the number of comparisons is

(*n* – 1) + (*n* – 2) + ... + 1

It can be shown that this is equal to ½*n*(*n* – 1) comparisons. So, for example, the number of comparisons that must be made to complete a selection sort on an array of 1000 elements is exactly

½ \* 1000 \* 999 = 499,500.

An important feature of a selection sort is that the number of comparisons is *always the same* for a given array length. It makes no difference, for example, if the array happens to be already sorted (see the last single stepper above). There are other sorting algorithms for which the initial ordering of the data can have a huge effect on the number of comparisons made.