An *insertion sort* is a common alternative to the selection sort. In this section, we demonstrate two different ways to implement it. The first requires the presence of a *destination array*, that is, an array of the same size as the array being sorted (the *source array*) into which the elements of the source array are inserted. When the process is complete, the source array is unchanged and the destination array contains the elements in order. Later, we will show how the algorithm can be modified so that it sorts an array *in place*, that is, without the need for a destination array.

Let's suppose we start with the source array shown below. The destination array is the same size and, to begin with, it is empty. We insert elements into the destination array as the algorithm proceeds.

Now we traverse the source array, starting at the first element. At each iteration, we copy an element from the source to the destination, inserting it so as to maintain the order of the elements of the destination array. In fact, a feature of the insertion sort algorithm is that, throughout the entire procedure, the destination array remains sorted.

The first element of the source array is 12. Since there is nothing in the destination array, we simply insert it there as the first element:

Next, we move on to the second element, 14, of the source array. Since we must insert 14 into the destination array so that it remains sorted, we insert 14 as the second element:

Next comes the third element, 15, of the source array. Since it is greater than 14, we insert it to the right of 14 in the destination array:

So far, nothing much seems to have changed. The start of the destination array is identical to the start of the source array. But now we turn our attention to the fourth element, 11, of the source array. We want to insert it into the destination array in such a way that the array remains sorted. The only way to achieve this is to insert it in front of 12. In order to create the necessary space at the beginning of the array, we must move the three elements we have inserted so far one place to the right:

This creates a space into which we copy 11:

The final element of the source array is 13. To keep the destination array sorted, we must insert 13 into that array between 12 and 14. We must therefore create the necessary space by moving the 14 and the 15 one place over to the right:

This creates a space into which we insert 13:

Notice that all elements of the source array have been copied into the destination array, and that the destination is sorted. The insertion sort is complete.

As you discovered in [Exercise 192](https://www.eimacs.com/eimacs/mainpage?epid=E201976902&cid=162149#Exe179), the number of comparisons made by an insertion sort depends on the initial ordering of the source array. If the source array is already sorted then *n* – 1 comparisons are made, where *n* is the length of the array. If the array is sorted in reverse order, then the number of comparisons is the same as for a selection sort, namely, ½*n*(*n* – 1). For any other ordering of the source array, the number of comparisons is somewhere between these two extremes.

The selection and insertion sort algorithms are called *quadratic sorts* and are said to execute in *quadratic time* because, if an array A2 is *c* times as long as an array A1 (where *c* is some positive number), then each of these sorting algorithms requires roughly *c*2 times as many comparisons to sort A2 as it does to sort A1. In particular, if A2 is 100 times as long as A1, then it will take a quadratic sort about 10,000 times as long to sort A2 as it will to sort A1.

In general, any process that involves an inner loop inside an outer loop, where both loops perform a number of iterations that is controlled by the value of the same variable, *n*, will run in quadratic time. For example, the execution time of this code:

  int n = <some integer>;   
  int i, j;   
  
  for ( i = 0 ; i < n ; i++ )   
  {   
    for ( j = 0 ; j < n ; j++ )   
    {   
       // code not shown   
    }   
  }

will be close to some fixed number of milliseconds multiplied by the square of the value of n.

In contrast, a process that involves a single loop like this:

  int n = <some integer>;   
  int i;   
  
  for ( i = 0 ; i < n ; i++ )   
  {   
    // code not shown   
  }

is said to run in *linear time* since the execution time will be close to some fixed number of milliseconds multiplied by the value of n. This is the case with an insertion sort when the source array is already sorted (or very nearly so). For such an array, an insertion sort requires roughly the same number of comparisons as there are elements in the array. Therefore, in the *best case*, an insertion sort runs in linear time.

**Exercise 193**

A programmer has written code that uses an insertion sort to sort an array into ascending order and then reports the amount of time taken. The code is executed on an array of 1,000 items that is already sorted into ascending order, and it takes 1.5 seconds. Approximately how long would it have taken if the array had initially been sorted into *descending* order?

[**Hint:** Since an insertion sort operates in linear time on an already-sorted array, you should be able to make an estimate of how long each comparison takes. Use this figure to help you estimate the answer to the question.]

### Exercise 193

An insertion sort of an already-sorted array of 1,000 elements requires 999 comparisons. The time taken is 1.5 seconds. So the time required for a single comparison is about 1.5 / 999 seconds. Applying the same sort algorithm to the same array if it were initially in reverse order would require ½ × 1000 × 999 comparisons. The time required would therefore be about

½ × 1000 × 999 × 1.5 / 999,

that is, 750 seconds.

Complete the following code to implement a class method insertionSort that takes as its only argument an Item[] source and that executes an insertion sort, returning a new Item[] with the elements sorted into ascending order.

public class Item implements Comparable<Item>   
{   
  private int myN;   
  
  public Item( int n )   
  {   
    myN = n;   
  }   
  
  public int compareTo( Item i )   
  {   
    return myN - i.getN();   
  }   
  
  public boolean equals( Object o )   
  {   
    // instanceof is not in the AP Java subset   
    return ( o instanceof Item && compareTo( (Item)o ) == 0 );   
  }   
  
  public String toString()   
  {   
    return "Item: " + myN;   
  }   
  
  public int getN()    
  {   
    return myN;   
  }   
  
  public static Item[] makeRandomItemArray( int len )   
  {   
    Random p = new Random();   
  
    Item[] a = new Item[ len ];   
    int i;   
    for ( i = 0 ; i < len ; i++ )   
      a[ i ] = new Item( p.nextInt( 100 ) - 50 );   
    return a;   
  }   
     
  public static void displayArray( Item[] array )   
  {   
    for ( Item item : array )   
      System.out.println( item );   
  }

 public static Item[] insertionSort( Item[] source )

﻿   {

     Item[] dest = new Item[ source.length ];

     for ( int i = 0 ; i < source.length ; i++ )

     {

       Item next = source[ i ];

       int insertindex = 0;

       int k = i;

       while ( k > 0 && insertindex == 0 )

       {

         if ( next.compareTo( dest[ k - 1 ] ) > 0 )

         {

           insertindex = k;

         }

         else

         {

           dest[ k ] = dest[ k - 1 ];

         }

         k--;

       }

       dest[ insertindex ] = next;

     }

     return dest;

   }

}   
    
public class MainClass   
{   
  public static void main( String[] args )   
  {

Item[] array = Item.makeRandomItemArray( 20 );

System.out.println( "Before:" );

Item.displayArray( array );

Item[] result = Item.insertionSort( array );

System.out.println( "\nAfter:" );

Item.displayArray( result );

  }   
}

Before:   
Item: -29   
Item: -22   
Item: 24   
Item: 36   
Item: 15   
Item: 22   
Item: -49   
Item: 5   
Item: -47   
Item: -27   
Item: 3   
Item: -1   
Item: -40   
Item: -32   
Item: -39   
Item: 40   
Item: 41   
Item: -17   
Item: 43   
Item: -11   
  
After:   
Item: -49   
Item: -47   
Item: -40   
Item: -39   
Item: -32   
Item: -29   
Item: -27   
Item: -22   
Item: -17   
Item: -11   
Item: -1   
Item: 3   
Item: 5   
Item: 15   
Item: 22   
Item: 24   
Item: 36   
Item: 40   
Item: 41   
Item: 43



In the next exercise, we have in mind to define a sortArrayList static method that can be used to sort an ArrayList of any kind of "sortable" objects. From our recent experience, we know that this means the objects to be sorted should all be instances of the same class and that class should implement an appropriate Comparable interface. So the question is: How can we tell the Java compiler that our method is prepared to accept an ArrayList of any kind of objects implementing a Comparable interface? The answer is to use a complicated-looking — and, to be honest, rather ugly — modification to the method header. In this case, the header will be

public static <T extends Comparable<T>> ArrayList<T> sortArrayList( ArrayList<T> source ).

Most aspects of this header are already familiar to you:

* The method's name is sortArrayList.
* It takes one argument of type ArrayList<T> (but without telling us what "T" is), for which the variable source will be used in the body of the method definition.
* It returns a value of the same type as the input (still without telling us what "T" is).
* It is modified in the usual way by the keywords public and static.

The unfamiliar part is the <T extends Comparable<T>> between the modifier keywords and the return data type. This is called a *type bound*; it is what actually tells us (and the Java compiler) what the mysterious "T" is. In English, it says, "T is any class that implements the corresponding Comparable<T> interface".

You do **not** have to know about type bounds for the Advanced Placement examination and you will never have to write headers of this kind in an examination context. In the few places where we use them in this online course, we will always provide them for you and we will make an appropriate explanatory comment.

Sorting an array in place has the advantage that no additional memory is consumed in the creation of a destination array. If the source array is very large, this may be a significant factor. On the other hand, there can be advantages to creating a destination array. For example, you may want to keep a copy of the unsorted array. Alternatively, if memory is not a problem but efficiency of performance may be, then you might use a destination structure — such as an ArrayList — that provides an efficient method of inserting an object.

It is interesting to note that, in our description, an insertion sort uses a sequential search (from right to left) to find the proper location for the next element. This sequential search is conducted on an array that is sorted (or, in the in-place case, on an array whose initial segment is sorted). It is therefore possible to use a binary search as we look for the proper location of the next element; such a sort is called a *binary insertion sort*. By employing a binary insertion sort algorithm, many fewer comparisons may be required, but we may not gain the full benefit of this saving unless we use a destination structure that allows efficient insertions.

A common variation of the insertion sort uses the break keyword. Let us demonstrate:

public class MainClass   
{   
  public static void insertionSort( int[] a )   
  {   
    int i, k, t;   
    for ( k = 2 ; k <= a.length ; k++ )   
    {   
      t = a[ k - 1 ];   
      for ( i = k - 1 ; i > 0 ; i-- )   
      {   
        if ( a[ i - 1 ] <= t )   
          break;   
        else   
          a[ i ] = a[ i - 1 ];   
       }   
       a[ i ] = t;   
     }   
  }   
  
  public static void main( String[] args )   
  {   
    int[] a = { 4, 2, 3, 9, 7, 8, 0, 1, 6, 5 };   
    insertionSort( a );   
    for ( int i : a )   
      System.out.print( i +  " " );   
  }   
}

0 1 2 3 4 5 6 7 8 9

Among other uses, the break keyword causes the currently executing loop to exit, with execution continuing at the first statement that follows the loop. Although the break keyword is *not* part of the Advanced Placement Computer Science Java subset, it is valuable for you to see an example of its use. In the above implementation of insertion sort, notice in particular how the current value of i is used to make the insertion. (What is the value of i if the loop exits normally?)