**Inserting In Order To Sort**

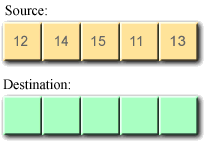
The **insertion sort (works by creating an empty array and then taking one element at a time from the original array and placing it into the new array; when you are done, the array is sorted)** algorithm involves creating an empty copy of the array you are sorting. Then, one by one, you take values from the array you want to sort and insert them into the empty array. Insertion gets its name from the fact that as you are inserting elements into the empty array, you look to see where the element should go and move array elements to the right as necessary.

### Part 1

The insertion sort works on the principle of finding the location a new value should be inserted within a sorted list. It is covered in detail in the next eIMACS lab.

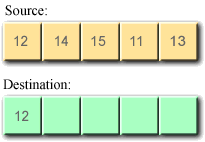
An *insertion sort* is a common alternative to the selection sort. In this section, we demonstrate two different ways to implement it. The first requires the presence of a *destination array*, that is, an array of the same size as the array being sorted (the *source array*) into which the elements of 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.

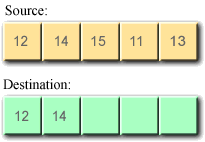


Now we traverse the source array, starting at the first element. At each iteration, we copy an element from the source to the destination, inserting it so as to maintain the order of the elements of the destination array. In fact, a feature of the insertion sort algorithm is that, throughout the entire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

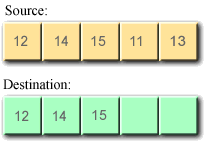
The first element of the source array is 12. Since there is nothing in the destination array, we simply insert it there as the first element:



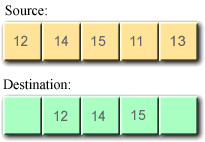
Next, we move on to the second element, 14, of the source array. Since we must insert 14 into the destination array so that it remains sorted, we insert 14 as the second element:



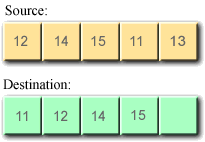
Next comes the third element, 15, of the source array. Since it is greater than 14, we insert it to the right of 14 in the destination array:



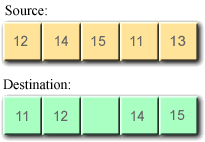
So far, nothing much seems to have changed. The start of the destination array is identical to the start of the source array. But now we turn our attention to the fourth element, 11, of the source array. We want to insert it into the destination array in such a way that the array remains sorted. The 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:



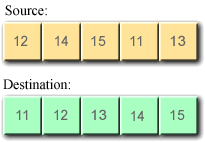
This creates a space into which we copy 11:



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.

What slows an insertion sort down is the need to slide elements of the destination array over to the right to make room for the insertion of the next element. It turns out that there is a very remarkable trick that can be employed in order to make this process more efficient. It relies on making comparisons with the current elements of the destination array *in reverse order*, starting with the last and working backward toward the first. Let us demonstrate.

Suppose the destination array currently contains the following elements:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 7 | 11 | 12 |  |  | … |

Further, suppose that the next element to be inserted is 10. To figure out where 10 should appear in the array, we start by considering the rightmost element of the destination array. After all, if 10 happens to be greater than or equal to this element, then we could save ourselves a lot of trouble by simply inserting 10 at the end of the array; there would be no need to slide any elements.

So we compare 10 with 12. Unfortunately, 10 is smaller than 12, so it must be the case that 10's place in the array is somewhere to the left of 12. It is at this point that the cleverness of the back-to-front comparison method is revealed. Since 10 is going to be inserted somewhere to the left of 12, it is certain that when the time comes to slide elements over, 12 will be one of the elements that has to be moved. *So we might as well move it now*:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 7 | 11 |  | 12 |  | … |

Continuing toward the left, the next element in the destination array is 11. Once again, 10 is smaller than 11, so it is certain that 11 will at some point have to be moved to the right to make room for 10. We therefore move it now:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 7 |  | 11 | 12 |  | … |

Continuing further to the left in the destination array, the next element is 7. Aha! 10 is greater than 7, so 10's place must be just to the right of 7. Since we have already created the space, we have only to drop 10 directly into it:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 7 | 10 | 11 | 12 |  | … |

We have inserted 10 into the destination array, the array remains sorted, and we may move on to the next element in the source array.

As you can see, the trick is to slide the elements of the destination array to the right while at the same time comparing each one with the element being inserted. This technique is employed in the code below. Single step through this code several times and notice that the existing elements of the destination array are compared with the next element to be inserted, starting with the rightmost existing element. If the next element is greater, then it is inserted and the inner loop ends. Otherwise, the element is moved one place to the right and the inner loop continues.

As you discovered in [Exercise 192](https://www.eimacs.com/eimacs/mainpage?epid=E2267326765&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.

The use of a destination array in our implementations of the insertion sort algorithm has made it easier for us to describe and investigate the algorithm, but any such implementation can easily be modified to perform an in-place sort — that is, a sort that operates directly on the source array.

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 +  " " );   
  }   
}

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?)

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

### Part 2

The Insertion sort is covered on the AP Exam, so it is worth your time to understand it well. You do not need to memorize the code from the standpoint of being able to re-write it, but you should be able to recognize an Insertion sort if you see it. You will need to be able to evaluate the output of the different sorts after each pass through of an array.

* Open the [07.09 Virtual Lecture Notes](https://lti.flvsgl.com/flvs-cat-content/l7jkq3lq7n9k2hh4dvq43i8ung/flvs-cat-session/apcomputersciencea_v20/module07/lesson09/pop/07_09b/07_09b_pop01.htm).
* Create a new project called 07.09 Insertion Sort in the Mod07 Lessons folder.
* Download the following Java files to the newly-created folder:
  + [HouseListing.java](https://lti.flvsgl.com/flvs-cat-content/l7jkq3lq7n9k2hh4dvq43i8ung/flvs-cat-session/apcomputersciencea_v20/module07/lesson09/docs/07_09b/HouseListing.java)
  + [TestListing2.java](https://lti.flvsgl.com/flvs-cat-content/l7jkq3lq7n9k2hh4dvq43i8ung/flvs-cat-session/apcomputersciencea_v20/module07/lesson09/docs/07_09b/TestListing2.java)
* Carefully review the information presented so you understand the insertion sort algorithm.