© 2011 by Douglas Wilhelm Harder. All rights reserved.
ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo
Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 7.02.
Assistances and comments will be acknowledged.

7.2 Insertion Sort

Suppose we have a list of size $k - 1$ that is already sorted. We can easily insert a kth new object into that list by starting at the back and moving items over until we find a location for it. For example, the list in Figure 1 has the first eight objects already sorted.

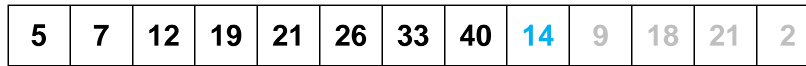

Figure 1. A sorted list of eight objects.

If we were to insert 14 into this sorted list of eight objects, we could would proceed backward: swap 14 and 40, then with 33, 26, 21, and 19. At this point, 12 < 14, so we are finished. This is quickly shown in Figure 2.

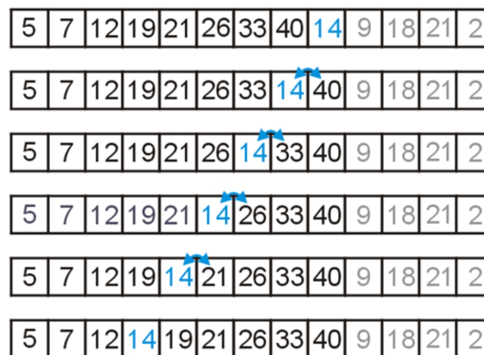

Figure 2. Inserting 14 into the sorted list of eight objects.

This creates a list of size 9. We can then proceed by inserting the next object, 9, into this sorted list.

7.2.1 The Insertion Sort Algorithm

The algorithm form insertion sort is:

1. Given a list of n items, treat the first item to be a sorted list of size 1.
2. Then, for k from 1 to $n - 1$:
   a. Insert the $(k + 1)^{\text{st}}$ object in the array into its appropriate location.
   b. This produces a list of $k + 1$ sorted objects.

After $n - 1$ steps, this produces a list of n sorted objects. This is clearly using the *insertion strategy* for sorting.

7.2.2 Implementation

Assuming we are trying to place the $(k + 1)^{\text{st}}$ object into the previously list of sorted entries. As soon as we find that the entry is in the correct location, we're finished—we can break out of the loop.

```
for ( int j = k; j > 0; --j ) {
    if ( array[j - 1] > array[j] ) {
        Object tmp = array[j];
        array[j] = array[j - 1];
        array[j - 1] = tmp;
    } else {
        // As soon as we don't need to swap, the (k + 1)st
        // is in the correct location
        break;
    }
}
```

This would be part of a larger function that would call this loop once for each value from $k = 1$ to $n - 1$:

```
template <typename Object>
void insertion_sort( Object *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                Object tmp = array[j];
                array[j] = array[j - 1];
                array[j - 1] = tmp;
            } else {
                // As soon as we don't need to swap, the (k + 1)st
               // is in the correct location
               break;
            }
        }
    }
}
```

7.2.3 Run-time Analysis

To do a run-time analysis, we will begin with the outer loop: k takes on the values 1, 2, 3 and so on until $k = n$ in which case, the condition fails. Therefore, the body of the outer for-loop will run $n - 1$ times with k taking the values 1 through $n - 1$.

The body of the inner loop contains an if-statement all components of which run in $\Theta(1)$ time. Thus, the body of the inner for-loop will run in $\Theta(1)$ time. As inner loop goes from k to 1, inclusive, the inner loop will run in $O(k)$. I use O instead of Θ because there is a break statement in the inner for-loop—the loop may terminate early.

In the worst case, however, the inner loop will run k times and therefore the worst-case run time will be

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \mathrm{O}(n^2).$$

Never-the-less, we should investigate the early termination of the loop. Suppose, for example, we pass insertion sort a sorted list. In this case, the inner condition `array[j - 1] > array[j]` will never be true: the run time of the inner loop will be $\Theta(1)$ and the run time of insertion sort will be $\Theta(n)$.

In fact, you will note that `array[j - 1] > array[j]` if and only if (a_{j-1}, a_j) forms an inversion. The swap corrects the inversion. Thus, the swap will only be performed for however many inversions occur within the list. If we represent the number of inversions by d, the run time is therefore $\Theta(n + d)$.

Thus, if we know asymptotically the number of inversions, we can make some definitive statements about the run time:

Inversions	Run Time of Insertion Sort	Comments
$d = \mathrm{O}(n)$	$\Theta(n)$	The list is sorted or almost sorted
$d = \omega(n)$	$\Theta(d)$	
$d = \Theta(n^2)$	$\Theta(n^2)$	Worst-case scenario

Also, we can make some trial runs and we note:

Array Size (n)	Approximate Run Time (ns)
8	175
16	750
32	2700
64	8000

Thus, it seems that approximately 10 instructions are being executed per inversion. Even sorting a list of size $n = 64$ takes only 8 μs. However, sorting a list of size $2^{23} \approx 8\,000\,000$ would require approximately one day—doubling the size of the list quadruples the required time. Later, we will see that an optimized quick sort algorithm will sort a list of this size in approximately 4 s.

Thus, we have the following runt times for insertion sort:

Inversions	Run Time	Comments
Worst	$\Theta(n^2)$	*E.g.,* reverse sorted
Average	$\Theta(n + d)$	Slow if $d = \omega(n)$
Best	$\Theta(n)$	Only if $d = \mathrm{O}(n)$

7.2.4 Optimizations

One optimization we can make is to observe that swapping,

```
Object tmp = array[j];
array[j] = array[j - 1];
array[j - 1] = tmp;
```

requires three assignments, while in reality, we could just assign the new object to a temporary variable and the place it into the correct position. In the following code fragment, we assign next object we are inserting into the list to `tmp` and we only place it into the array when we find its position.

```
template <typename Object>
void insertion( Object *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        Object tmp = array[k];

        for ( int j = k; k > 0; --j ) {
            if ( array[j - 1] > tmp ) {
                array[j] = array[j - 1];
            } else {
                 array[j] = tmp;
                 goto finished;
            }
        }

        array[0] = tmp;  // only executed if tmp < array[0]
        finished: ;  // empty statement
    }
}
```

Notice that if the new object is less than `array[0]`, we need to insert tmp into that location after the end of the loop. This is only necessary in this special case.

7.2.5 Goto?!

Most programmers have been told the religious edict that *using goto is bad*. This is, in general, a very good piece of advice. However, in this case, the behaviour is clear and there is no other elegant technique for dealing with this problem. There are alternate solutions, but it either involves the use of a flag or requires an implicit assumption.

For example, we could make the following check at the end

```
template <typename Object>
void insertion( Object *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        Object tmp = array[k];

        for ( int j = k; k > 0; --j ) {
            if ( array[j - 1] > tmp ) {
                array[j] = array[j - 1];
            } else {
                 array[j] = tmp;
                 break;
            }
        }

        // If tmp < array[0], the entire loop would finish
        // without some entry array[j] being assigned.  This means
        // the object must be inserted at the end.

        if ( array[0] > tmp ) {
            array[0] = tmp;
        }
    }
}
```

however, the result is non-intuitive. In this case, the use of a `goto` is in fact more intuitive. The crusade against the `goto` statement was start with Edsger Dijkstra's 1968 commentary "Go To Statement Considered Harmful". Providing a more balanced view was Donald Knuth's more-insightful paper "Structured Programming with go to Statements". Unfortunately, absolute bans are always easier to enforce than carefully reasoning.