LINEAR INSERTION SORT

Textbook (pages 95 – 103)

- The term *insertion sort* refers to an entire category of sorting algorithms which work by successively taking each item in the array and finding its proper position.

- The particular insertion sort covered in your textbook is called *linear insertion sort*.

- Assume you have an array in which all items in positions 0 through \texttt{first-1} are sorted; those at positions \( \geq \texttt{first} \) remain unsorted.

  ![Diagram of linear insertion sort](image)

  - Linear insertion works by taking the item at position \texttt{first} and finding its proper position among the sorted items in positions \( \ < \texttt{first} \). Larger items are shifted up one position.
The item being considered is placed into its proper position so that items 0 through \textit{first} are now in order.

\textit{first} is incremented and the process continues until the last item has been inserted into place.

Loop invariant of linear insertion (page 103): all data items at positions < \textit{first} are sorted.

Implementation in Java is on page 101.
```java
public void linearInsertionSort( double[] list )
// Precondition: list is not null.
// Postcondition: the items in list are in ascending order.
{
    // first marks the first unsorted item in the list
    // items in positions 0 to first-1 are kept sorted
    int first, left;
    double temp;
    for ( first = 1; first < list.length; first++ )
    { // list[0] to list[first-1] are sorted
        // save the unsorted item into a temporary
        temp = list[first];
        // start with the item immediately "left" of first
        left = first-1;
        // shifting items that are > temp
        while ( left >= 0 && list[left] > temp )
        { // shift it "right"
            list[left+1] = list[left];
            left--;
        }
        // either left == -1 || list[left] <= temp
        // so temp goes in the position to it's "right"
        list[left+1] = temp;
        // list[0] to list[first] are now sorted
    }
}
```
Analysis of Linear Insertion Sort (page103)

Linear insertion sort makes \(n-1\) passes on an array of size \(n\). On the first pass, it compares \(a[1]\) with \(a[0]\) and puts them in order, requiring 1 comparison and up to 1 shift. The second pass considers \(a[2]\) and finds its proper place within \(a[0]\) through \(a[1]\), requiring up to 2 comparisons and up to 2 shifts. The third pass considers \(a[3]\) and finds its proper place within \(a[0]\) through \(a[2]\), requiring up to 3 comparisons and up to 3 shifts. This continues until all items except \(a[n-1]\) are in place. For the final pass, linear insertion places this item within \(a[0]\) through \(a[n-2]\), taking up to \(n-1\) comparisons and \(n-1\) shifts.

The total number of computation steps is, at worst case, \(\sum_{k=1}^{n-1} k\) comparisons and \(\sum_{k=1}^{n-1} k\) swaps, totaling:

\[
1+2+\ldots+(n-1)+ 1+2+\ldots+(n-1)
\]

\[
= \left( \frac{(n-1)n}{2} \right) + \left( \frac{(n-1)n}{2} \right)
\]

\[
= \left( \frac{n^2}{2} + \frac{n}{2} \right) + \left( \frac{n^2}{2} + \frac{n}{2} \right)
\]

\[
= n^2 + n
\]

Thus, linear insertion is an \(O(n^2)\) sort, where \(n\) is the size of the array.

Linear insertion sort runs quickly if the array is already sorted. In this case it makes \(n-1\) passes on an array of size \(n\). On the first pass, it compares \(a[1]\) with \(a[0]\) and sees that \(a[1]\) is already > \(a[0]\) with 1 comparison and no shifts. The second pass considers \(a[2]\) and finds it > \(a[1]\) with 1 comparison and no shifts. The third pass finds \(a[3]\) in its proper position with 1 comparison and no shifts. And so it goes to the end of the array.

Thus, the total number of computation steps is \(\sum_{k=1}^{n-1} 1\) comparisons, totaling:

\[n-1\]

When given an \(n\)-element array is in random order, the linear insertion sort runs in \(O(n^2)\) time. When the array is already sorted, linear insertion runs in \(O(n)\) time. The more sorted the array is, the more closely its time is to \(O(n)\).

In many applications you can exploit this property to improve performance. For example, consider a point-of-sale cash register in a video store connected to an application that maintains the list of videos, either available or checked out. There are thousands of titles in the list – sorted so that binary search can be used to find items in the list. Every week a dozen or so new videos are released. The application can efficiently add them to the list by appending them to the end.
and using linear insertion to reorder the list. Since the list is mostly ordered (except for the few new titles) the process can be accomplished very close to $O(n)$ time.