insertion sort

Canonical name	InsertionSort
Date of creation	2013-03-22 11:44:38
Last modified on	2013-03-22 11:44:38
Owner	mathcam (2727)
Last modified by	mathcam (2727)
Numerical id	16
Author	mathcam (2727)
Entry type	Algorithm
Classification	msc 68P10
Classification	msc 55U40
Classification	msc 55U99
Classification	msc 55U15
Classification	msc 55U10
Classification	msc 55P20
Classification	msc 55-00
Classification	msc 85-00
Classification	msc 83-02
Related topic	SortingProblem
Related topic	BinarySearch
Related topic	SelectionSort

The Problem

See the sorting problem.

The Algorithm

Suppose $L = \{x_1, x_2, \ldots, x_n\}$ is the initial list of unsorted elements. The *insertion sort* algorithm will construct a new list, containing the elements of L in order, which we will call L'. The algorithm constructs this list one element at a time.

Initially L' is empty. We then take the first element of L and put it in L'. We then take the second element of L and also add it to L', placing it before any elements in L' that should come after it. This is done one element at a time until all n elements of L are in L', in sorted order. Thus, each step i consists of looking up the position in L' where the element x_i should be placed and inserting it there (hence the name of the algorithm). This requires a search, and then the shifting of all the elements in L' that come after x_i (if L' is stored in an array). If storage is in an array, then the binary search algorithm can be used to quickly find x_i's new position in L'.

Since at step i, the length of list L' is i and the length of list L is $n - i$, we can implement this algorithm as an in-place sorting algorithm. Each step i results in $L[1..i]$ becoming fully sorted.

Pseudocode

This algorithm uses a modified binary search algorithm to find the position in L where an element *key* should be placed to maintain ordering.

Algorithm INSERTION_SORT(L, n)
Input: A list L of n elements
Output: The list L in sorted order

begin
  for $i \leftarrow 1$ to n **do**
    begin
      $value \leftarrow L[i]$
      $position \leftarrow Binary\_Search(L, 1, i, value)$
      for $j \leftarrow i$ downto $position$ **do**
        $L[j] \leftarrow L[j-1]$
      $L[position] \leftarrow value$
    end

```
end

function Binary_Search(L, bottom, top, key)
begin
    if bottom >= top then
        Binary_Search ← bottom
    else
        begin
            middle ← (bottom + top)/2
            if key < L[middle] then
                Binary_Search ← Binary_Search(L, bottom, middle − 1, key)
            else
                Binary_Search ← Binary_Search(L, middle + 1, top, key)
        end
end
```

Analysis

In the worst case, each step i requires a shift of $i-1$ elements for the insertion (consider an input list that is sorted in reverse order). Thus the runtime complexity is $\mathcal{O}(n^2)$. Even the optimization of using a binary search does not help us here, because the deciding factor in this case is the insertion. It is possible to use a data type with $\mathcal{O}(\log n)$ insertion time, giving $\mathcal{O}(n \log n)$ runtime, but then the algorithm can no longer be done as an in-place sorting algorithm. Such data structures are also quite complicated.

A similar algorithm to the insertion sort is the selection sort, which requires fewer data movements than the insertion sort, but requires more comparisons.