

Sorting

Summary: We consider a variety of techniques used to put a list or vector in order, using a binary comparison operation to determine the ordering of pairs of elements.

Contents:

- The Problem of Sorting
- The Insertion Sort Algorithm
 - Inserting Elements
 - Inserting Elements, Revisited
 - Insertion Sorting, Continued
- Sorting a Vector

The Problem of Sorting

Sorting a collection of values -- arranging them in a fixed order, usually alphabetical or numerical -- is one of the most common computing applications. When the number of values is even moderately large, sorting is such a tiresome, error-prone, and time-consuming process for human beings that the programmer should automate it whenever possible. For this reason, computer scientists have studied this application with extreme care and thoroughness.

One of the clear results of their investigations is that no one algorithm for sorting is best in all cases. Which approach is best depends on whether one is sorting a small collection or a large one, on whether the individual elements occupy a lot of storage (so that moving them around in memory is time-consuming), on how easy or hard it is to compare two elements to figure out which one should precede the other, and so on. In this course we'll be looking at two of the most generally useful algorithms for sorting: *insertion sort*, which is the subject of this reading, and *merge sort*, which we'll talk about in another reading. In our general exploration of sorting, we may also discuss other sorting algorithms.

Imagine first that we're given a collection of values and a rule for arranging them. The values might actually be stored either in a list or in a vector; let's assume first that they are in a list. The rule for arranging them typically takes the form of a predicate with two parameters that can be applied to any two values in the set to determine whether the first of them could precede the second when the values have been sorted. (For example, if one wants to sort a set of real numbers into ascending numerical order, the rule should be the predicate \leq ; if one wants to sort a set of strings into alphabetical order, ignoring case, the rule should be `string-ci<=?`, and so on.)

The Insertion Sort Algorithm

Insertion sort works by taking the values one by one and inserting each one into a new list that it constructs, constantly maintaining the condition that the elements of the new list are in the desired order with respect to one another. Clearly, this condition will not be maintained if each element is added to the

new list at the beginning, using `cons`; instead, the insertion sort algorithm adds each element at a carefully selected position within the new list, placing the new element *after* each previously placed element that precedes it according to the given precedence rule, but *before* every such element that it precedes. The following procedure, `insert-number`, adds a new element to a list in exactly this way. For the moment, we'll assume that the elements of the list are real numbers and that we want to sort them into ascending order; `<=` is therefore used as the ordering predicate.

Inserting Elements

We begin with the procedure used to insert a new value into a list that is already in order.

```

;;; Procedure:
;;;   insert-number
;;; Parameters:
;;;   new-element, a real numbers
;;;   sorted, a list of real numbers
;;; Purpose:
;;;   Insert new-element into sorted.
;;; Produces:
;;;   new-ls, a new list of real numbers
;;; Preconditions:
;;;   sorted is a list of numbers arranged in increasing order.   That is,
;;;   (<= (list-ref sorted i) (list-ref sorted (+ i 1)))
;;;   for all reasonable values of i. [Unverified]
;;;   new-element is a number. [Unverified]
;;; Postconditions:
;;;   new-ls is a list of numbers arranged in increasing order.
;;;   new-ls is a permutation of (cons new-element sorted).
(define insert-number
  (lambda (new-element sorted)
    (cond ((null? sorted) (list new-element))
          ((<= new-element (car sorted)) (cons new-element sorted))
          (else (cons (car sorted) (insert-number new-element (cdr sorted)))))))

```

In English: If the list into which the new element is to be inserted is empty, return a list containing only the new element. If the new element can precede the first element of the existing list, then, since the existing list is assumed to be sorted already, it must also be able to precede *every* element of the existing list, so attach the new element onto the front of the existing list and return the result. Otherwise, we haven't yet found the place, so issue a recursive call to insert the new element into the `cdr` of the current list, then reattach its `car` at the beginning of the result.

Inserting Elements, Revisited

The preceding version of the `insert-number` procedure is not tail-recursive. When dealing with long lists, you may want to use the following tail-recursive version, which uses memory more economically. (A “tail-recursive” procedure is one in which the recursive call is not followed by additional work; implementations of Scheme use less memory for tail recursion than they use for recursion in which the recursive call is followed by additional work.)

```
(define insert-number2
  (lambda (new-element sorted)
    (let kernel ((rest sorted)
                 (bypassed null))
      (cond ((null? rest) (reverse (cons new-element bypassed)))
            ((<= new-element (car rest))
             (append (reverse (cons new-element bypassed)) rest))
            (else (kernel (cdr rest) (cons (car rest) bypassed)))))))
```

Of course, our lists typically have more than just numbers in them. In the associated laboratory you will experiment with generalized forms of `insert`. How do we generalize? We make the operation that compares values a parameter to the procedure. We typically call that procedure `may-precede?`. Here's one possible version of `insert` that accepts the additional parameter.

```
(define insert
  (lambda (new-value sorted may-precede?)
    (let kernel ((rest sorted)
                 (bypassed null))
      (cond ((null? rest) (reverse (cons new-value bypassed)))
            ((may-precede? new-value (car rest))
             (append (reverse (cons new-value bypassed)) rest))
            (else (kernel (cdr rest) (cons (car rest) bypassed)))))))
```

Insertion Sorting, Continued

Now let's return to the overall process of sorting an entire list. The insertion sort algorithm simply takes up the elements of the list to be sorted one by one and inserts each one into a new list, initially empty:

```
;;; Procedure:
;;; insertion-sort-numbers
;;; Parameters:
;;; numbers, a list of real numbers
;;; Purpose:
;;; Sorts numbers
;;; Produces:
;;; sorted, a list of real numbers
;;; Preconditions:
;;; (none)
;;; Postconditions:
;;; sorted is a list of real numbers.
;;; sorted is organized in increasing order. That is,
;;; (list-ref sorted i) is less than or equal to
;;; (list-ref sorted (+ i 1)) for all reasonable values
;;; of i.
;;; sorted is a permutation of numbers.
(define insertion-sort-numbers
  (lambda (numbers)
    (let helper ((unsorted numbers) ; The remaining unsorted values
                 (sorted null)) ; The sorted values
      (if (null? unsorted)
          sorted
          (helper (cdr unsorted) (insert-number (car unsorted) sorted))))))
```

Sorting a Vector

Finally, let's consider the rather different case in which the values that we want to arrange are presented as a vector and the goal of the sorting algorithm is to *overwrite* the old arrangement of those values with a new, sorted arrangement of the same values. This type of sorting is often called *in-place* sorting.

Instead of constructing a new vector, we partition the original vector into two subvectors: a sorted subvector, in which all of the elements are in the correct order relative to one another, and an unsorted subvector in which the elements are still in their original positions. The two subvectors are not actually separated; instead, we just keep track of a boundary between them inside the original vector. Items to the left of the boundary are in the sorted subvector; items to its right, in the unsorted one. Initially the boundary is at the left end of the vector. The plan is to shift it, one position at a time, to the right end. When it arrives, the entire vector has been sorted.

Here's the plan for the main algorithm.

```
;;; Procedure:
;;; insertion-sort!
;;; Parameters:
;;; may-precede?, a binary predicate
;;; vec, a vector
;;; Purpose:
;;; Sorts the vector.
;;; Produces:
;;; [Nothing; sorts in place]
;;; Preconditions:
;;; vec is a vector.
;;; may-precede? can be applied to any two elements of vec.
;;; may-precede? is transitive.
;;; Postconditions:
;;; The final state of vec is a permutation of the original state.
;;; vec is sorted. That is,
;;; (may-precede? (vector-ref vec i) (vector-ref vec (+ i 1)))
;;; for all reasonable values of i.
(define insertion-sort!
  (lambda (may-precede? vec)
    (let ((len (vector-length vec)))
      (let kernel ((boundary 1)) ; The index of the first unsorted value
        (if (< boundary len) ; If we have elements left to sort
            (begin
              (insert! (vector-ref vec boundary)
                       vec
                       boundary
                       may-precede?)
              (kernel (+ boundary 1))))))))))
```

The `insert!` procedure takes four parameters: an element to be inserted into the sorted part of the vector, the vector itself, the current boundary position, and the comparison procedure. The new element can be inserted at any position up to and including the current boundary position, but it must be placed in the correct order relative to elements to the left of that boundary. This means that any elements that should follow the new one should be shifted one position to the right in order to make room for the new one. (Elements that precede the new one can keep their current positions.)

```

;;; Procedure:
;;; insert!
;;; Parameters:
;;; new-element, a value
;;; vec, a vector of values
;;; boundary, an index into the vector
;;; may-precede?, a binary predicate
;;; Purpose:
;;; Insert new-element into the portion of vec between 0 and
;;; boundary, inclusive.
;;; Produces:
;;; Nothing; called for side effects.
;;; Preconditions:
;;; 0 <= boundary < (vector-length vec)
;;; The elements in positions 0..boundary-1 of vec are sorted.
;;; That is, (may-precede? (vector-ref vec i) (vector-ref vec (+ i 1)))
;;; for all 0 < i < boundary-2.
;;; Postconditions:
;;; The elements in positions 0..boundary of vec are sorted.
;;; That is, (may-precede? (vector-ref vec i) (vector-ref vec (+ i 1)))
;;; for all 0 < i < boundary.
;;; The elements in positions 0..boundary of vec after insert! finishes
;;; are a permutation of new-element and the elements that were in
;;; positions 0..(boundary-1) before the procedure started.
(define insert!
  (lambda (new-element vec boundary may-precede?)
    (let kernel ((pos boundary))
      (cond
        ; If we've reached the left end of the vector, we've run out of
        ; elements to shift. Insert the new element.
        ((zero? pos)
         (vector-set! vec pos new-element))
        ; If we've reached a point at which the element to the left
        ; is smaller, we insert the new element here.
        ((may-precede? (vector-ref vec (- pos 1)) new-element)
         (vector-set! vec pos new-element))
        ; Otherwise, we shift the current element to the right and
        ; continue.
        (else
         (vector-set! vec pos (vector-ref vec (- pos 1)))
         (kernel (- pos 1)))))))

```

How does this work? We assume that there's a "space" at position `pos` of the vector. (That is, that we can safely insert something there without removing anything from the vector.) We know that the condition holds at the beginning from the description. That is, the postcondition specifically ignores the value that was in the boundary position. We also know that the conditional holds from the way `insert!` was called from `insertion-sort!`, since the boundary is initially the position of the value we insert.

Now, what do we do? If the position is at the left end of the vector, there's nothing smaller in the vector, so we just put the new value there. If the thing to the left of the current position is smaller, we know we've reached the right place, so we put the value there. In every other case, the value to the left is larger than the value we want to insert, so we shift that value right (into the `pos` position) and continue working one position to the left. Since we've copied the value to the right, it remains safe to insert something in the position just vacated (that is, `pos-1`).

You will have a chance to explore this procedure further in the laboratory.

Copyright © 2007 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.