

Sorting

- 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 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 discussion and *merge sort*, which we'll talk about in another reading.

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 typically takes the form of a predicate of arity 2 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 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; \leq 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 number
;;; ls, a list of numbers
;;; Purpose:
;;; Insert new-element into ls.
;;; Produces:
;;; new-ls, a new list of numbers
;;; Preconditions:
;;; ls is a list of numbers arranged in increasing order.
;;; That is, (list-ref ls i) is less than or equal to
;;; (list-ref ls (+ 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.
;;; Each element of ls appears in new-ls.
;;; new-element appears in new-ls.
;;; No other elements appear in new-ls.
(define insert-number
  (lambda (new-element ls)
    (cond ((null? ls) (list new-element))
          ((<= new-element (car ls)) (cons new-element ls))
          (else (cons (car ls) (insert-number new-element (cdr ls)))))))
```

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 space more economically:

```
(define insert-number2
  (lambda (new-element ls)
    (let kernel ((rest ls)
                 (bypassed null))
      (cond ((null? rest) (revappend bypassed (list new-element)))
            ((<= new-element (car rest))
             (revappend bypassed (cons new-element rest)))
            (else (kernel (cdr rest) (cons (car rest) bypassed)))))))

;;; Procedure:
;;; revappend
```

```

;;; Parameters:
;;; left, a list
;;; right, a list
;;; Purpose:
;;; Join the reverse of left to right.
;;; Produces:
;;; newlst, a list
;;; Preconditions:
;;; left and right are lists. [Unverified]
;;; Postconditions:
;;; newlst is the same as (append (reverse left) right).
(define revappend
  (lambda (left right)
    (if (null? left)
        right
        (revappend (cdr left) (cons (car left) right)))))

```

Of course, our lists typically have more than just numbers in them. In the associated laboratory you will experiment with generalized forms of insert. Here's one possibility:

```

(define insert
  (lambda (new-value lst may-precede?)
    (let kernel ((rest lst)
                (bypassed null))
      (cond ((null? rest) (revappend bypassed (list new-value)))
            ((may-precede? new-value (car rest))
             (revappend bypassed (cons new-value 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 numbers
;;; Purpose:
;;; Sorts numbers
;;; Produces:
;;; sorted, a list of numbers
;;; Preconditions:
;;; numbers is a list of numbers. [Unverified]
;;; Postconditions:
;;; sorted is a list of 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 and numbers contain the same values. That is,
;;; no numbers were added or removed in the construction of
;;; sorted.
(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.

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.
;;; Preconditions:
;;;   vec is a vector.
;;;   may-precede? can be applied to any two elements of vec.
(define insertion-sort-numbers!
  (lambda (may-precede? vec)
    (let ((len (vector-length vec)))
      (let helper ((boundary 0)) ; 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?)
              (helper (+ boundary 1))))))))))

```

The `insert!` procedure takes four arguments: 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.)

You will write this insert! procedure in the lab.