

Merge Sort

Summary: In a past reading and the corresponding laboratory, we've explored the basics of sorting using insertion sort. In this reading, we turn to another sorting algorithm, *merge sort*.

The Costs of Insertion Sort

In looking at algorithms, we often ask ourselves how many “steps” the algorithm typically uses. Let's try to look at how much effort the insertion sort algorithm expends in sorting a list of n values, starting from a random initial arrangement. Recall that insertion sort uses two lists: a growing collection of sorted values and a shrinking collection of values left to examine. At each step, it inserts a value into the collection of sorted values.

In general, insertion sort has to look through half of the elements in the sorted part of the data structure to find the correct insertion point for each new value it places. The size of that sorted part increases linearly from 0 to n , so its average size is $n/2$ and the average number of comparisons needed to insert one element is $n/4$. Taking all the insertions together, then, the insertion sort performs about $n^2/4$ comparisons to sort the entire set. That is, we do an average of $n/4$ work n times, giving $n^2/4$.

This function grows much more quickly than the size of the input list. For example, if we have 10 elements, we do about 25 comparisons. If we have 20 elements, we do about 100 comparisons. If we have 40 elements, we do about 400 comparisons. And, if we have 100 elements, we do about 2500 comparisons.

Accordingly, when the number of values to be sorted is large (greater than one thousand, say), it is preferable to use a sorting method that is more complicated to set up initially but performs fewer comparisons per value in the list.

As we saw in the case of binary search, it is often profitable to divide an input in half. For binary search, we could throw away half and then recurse on the other half. Clearly, for merging, we cannot throw away part of the list. However, we can still rely on the idea of dividing in half. That is, we'll divide the list into two halves, sort them, and then do something with the two result lists.

Here's a sketch of the algorithm in Scheme:

```
;;; Procedure:
;;; merge-sort
;;; Parameters:
;;; stuff, a list to sort
;;; may-precede?, a binary predicate that compares values.
;;; Purpose:
;;; Sort stuff.
;;; Produces:
;;; sorted, a sorted list
;;; Preconditions:
;;; may-precede? can be applied to any two values in stuff.
;;; may-precede? represents a transitive operation.
```

```

;;; Postconditions:
;;; The result list is sorted. That is, the key of any
;;; element may precede the key of any subsequent element.
;;; In Scheme, we'd say that
;;; (may-precede? (list-ref sorted i)
;;; (list-ref sorted (+ i 1)))
;;; holds.
;;; sorted and stuff have the same elements (although potentially
;;; in different orders).
;;; Does not affect stuff.
;;; sorted may share cons cells with stuff.
(define merge-sort
  (lambda (stuff may-precede?)
    ; If there are only zero or one elements in the list,
    ; the list is already sorted.
    (if (or (null? stuff) (null? (cdr stuff)))
        stuff
        ; Otherwise, split the list in half
        (let* ((halves (split stuff))
              (firsthalf (car halves))
              (secondhalf (cadr halves)))
          ; Sort each half.
          (let* ((sortedfirst (merge-sort firsthalf))
                (sortedsecond (merge-sort secondhalf)))
            ; Do some more stuff
            ??))))))

```

Merging

But what do we do once we've sorted the two sublists? We need to put them back into one list. Through habit, we refer to the process of joining two sorted lists as *merging*. It is relatively easy to merge two lists: You repeatedly take the smallest remaining element of either list. When do you stop? When you run out of elements in one of the lists, in which case you use the elements of the remaining list. Putting it all together, we get the following:

```

;;; Procedure:
;;; merge
;;; Parameters:
;;; sorted1, a sorted list.
;;; sorted2, a sorted list.
;;; may-precede?, a binary predicate that compares keys
;;; Purpose:
;;; Merge the two lists.
;;; Produces:
;;; sorted, a sorted list.
;;; Preconditions:
;;; may-precede? can be applied to any two values from
;;; sorted1 and/or sorted2.
;;; may-precede? represents a transitive operation.
;;; sorted1 and sorted2 are sorted.
;;; Postconditions:
;;; The result list is sorted.
;;; Every element in sorted1 appears in sorted.
;;; Every element in sorted2 appears in sorted.
;;; Every element in sorted appears in sorted1 or sorted2.

```

```

;;; Does not affect sorted1 or sorted2.
;;; sorted may share cons cells with sorted1 or sorted2.
(define merge
  (lambda (sorted1 sorted2 may-precede?)
    (cond
      ; If the first list is empty, return the second
      ((null? sorted1) sorted2)
      ; If the second list is empty, return the first
      ((null? sorted2) sorted1)
      ; If the first element of the first list is smaller,
      ; make it the first element of the result and recurse.
      ((may-precede? (car sorted1) (car sorted2))
       (cons (car sorted1)
              (merge (cdr sorted1) sorted2 may-precede?)))
      ; Otherwise, do something similar using the first element
      ; of the second list
      (else
       (cons (car sorted2)
              (merge sorted1 (cdr sorted2) may-precede?)))))))

```

Splitting

All that we have left to do is to figure out how to split a list into two parts. One easy way is to get the length of the list and then cdr down it for half the elements, accumulating the skipped elements as you go. Since it's easiest to accumulate a list in reverse order, we re-reverse it when we're done.

```

;;; Procedure:
;;; split
;;; Parameters:
;;; lst, a list
;;; Purpose:
;;; Split a list into two nearly-equal halves.
;;; Produces:
;;; halves, a list of two lists
;;; Preconditions:
;;; lst is a list.
;;; Postconditions:
;;; halves is a list of length two.
;;; Each element of halves is a list (which we'll refer to as
;;; firsthalf and secondhalf.
;;; Every element in the original list is in exactly one of the
;;; firsthalf and secondhalf.
;;; No other elements are in firsthalf or secondhalf.
;;; Does not modify lst.
;;; Either firsthalf or secondhalf may share cons cells with lst.
(define split
  (lambda (lst)
    ;;; helper
    ;;; Remove the first count elements of a list. Return the
    ;;; pair consisting of the removed elements (in order) and
    ;;; the remaining elements.
    (let helper ((remaining lst) ; Elements remaining to be used
                  (revacc null) ; Accumulated initial elements
                  (count 0) ; How many elements left to use
                  (quotient (length lst) 2)))

```

```
; If no elements remain to be used,
(if (= count 0)
    ; The first half is in revacc and the second half
    ; consists of any remaining elements.
    (list (reverse revacc) remaining)
    ; Otherwise, use up one more element.
    (helper (cdr remaining)
            (cons (car remaining) revacc)
            (- count 1))))))
```

In the corresponding lab, you'll have an opportunity to consider other ways to split the list. In that lab, you'll work with a slightly changed version of the code that identifies a "key" for each value in the list and then compares keys.