

Merge Sort

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

Contents:

- The Costs of Insertion Sort
- Divide and Conquer
- Merging
- Splitting
- An Alternative: From Small Lists to Large Lists
- The Costs of Merge Sort
- Documenting Merge Sort

The Costs of Insertion Sort

In looking at algorithms, we often ask ourselves how many “steps” the algorithm typically uses. Rather than looking at every kind of step, we tend to focus on particular kinds of steps, such as the number of times we have to call `vector-set!` or the number of values we look at.

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.

Divide and Conquer

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:

```
(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 values.
;;; 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. (Merge sort doesn't really care that they're in the original order, but perhaps we want to use `split` for other purposes.)

```

;;; 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).
;;;   lst is a permutation of (append firsthalf secondhalf).
;;;   The lengths of firsthalf and secondhalf differ by at most 1.
;;;   Does not modify lst.
;;;   Either firsthalf or secondhalf may share cons cells with lst.
(define split
  (lambda (lst)
    ;;; kernel
    ;;; Remove the first count elements of a list. Return the
    ;;; pair consisting of the removed elements (in order) and
    ;;; the remaining elements.
    (let kernel ((remaining lst) ; Elements remaining to be used
                  (revacc null) ; Accumulated initial elements
                  (count      ) ; 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.
    (kernel (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.

An Alternative: From Small Lists to Large Lists

There's an awful lot of recursion going on in merge sort as we repeatedly split the list again and again and again until we reach lists of length one. Rather than doing all that recursion, we can start by building all the lists of length one and then repeatedly merging pairs of neighboring lists. For example, suppose we start with sixteen values, each in a list by itself.

```
((20) (42) (35) (10) (69) (92) (77) (27) (67) (62) (1) (66) (5) (45) (25) (90))
```

When we merge neighbors, we get sorted lists of two elements. At some places such as when we merge (20) and (42), the elements stay in their respective order. At other places, such as when we merge (35) and (10), we need to swap order to build ordered lists of two elements.

```
((20 42) (10 35) (69 92) (27 77) (62 67) (1 66) (5 45) (25 90))
```

Now we can merge these sorted lists of two elements into sorted lists of four elements.

```
((10 20 35 42) (27 69 77 92) (1 62 66 67) (5 25 45 90))
```

We can merge these sorted lists of four elements into sorted lists of eight elements.

```
((10 20 27 35 42 69 77 92) (1 5 25 45 62 66 67 90))
```

Finally, we can merge these sorted lists of eight elements into one sorted list of sixteen elements.

```
((1 5 10 20 25 27 35 42 45 62 66 67 69 77 90 92))
```

Translating this technique into code is fairly easy. We use one helper, `merge-pairs` to merge neighboring pairs. We use a second helper, `repeat-merge` to repeatedly call `merge-pairs` until there are no more pairs to merge.

```

(define new-merge-sort
  (lambda (stuff may-precede?)
    (letrec (
      ; Merge neighboring pairs in a list of lists
      (merge-pairs
       (lambda (list-of-lists)
         (cond

```

```

      ; Base case: Empty list.
      ((null? list-of-lists) null)
      ; Base case: Single-element list (nothing to merge)
      ((null? (cdr list-of-lists)) list-of-lists)
      ; Recursive case: Merge first two and continue
      (else (cons (merge (car list-of-lists) (cadr list-of-lists)
                        may-precede?)
                  (merge-pairs (caddr list-of-lists))))))
; Repeatedly merge pairs
(repeat-merge
 (lambda (list-of-lists)
  ; Show what's happening
  ; (write list-of-lists) (newline)
  ; If there's only one list in the list of lists
  (if (null? (cdr list-of-lists))
      ; Use that list
      (car list-of-lists)
      ; Otherwise, merge neighboring pairs and start again.
      (repeat-merge (merge-pairs list-of-lists))))))
(repeat-merge (map list stuff))))

```

The Costs of Merge Sort

At the beginning of this reading, we saw that insertion sort takes approximately $n^2/4$ steps to sort a list of n elements. How long does merge sort take? We'll look at `new-merge-sort`, since it's easier to analyze. However, since it does essentially the same thing as the original `merge-sort`, just in a slightly different order, the running time will be similar.

We'll do our analysis in a few steps. First, we will consider the number of steps in each call to `merge-pairs`. Next, we will consider the number of times `repeat-merge` calls `merge-pairs`. Finally, we'll put the two together. To make things easier, we'll assume that n (the number of elements in the list) is a power of two.

Initially, `repeat-merge` calls `merge-pairs` on n lists of length 1 to merge them into $n/2$ lists of length 2. Building a list of length 2 takes approximately two steps, so `merge-pairs` takes approximately n steps to do its first set of merges.

Next, `repeat-merge` calls `merge-pairs` on $n/2$ lists of length 2 to merge them into $n/4$ lists of length 4. Building a merged list of length 4 takes approximately four steps, so `merge-pairs` takes approximately n steps to build $n/4$ list of length 4.

Each time, `repeat-merge` calls `repeat-merge` to merge $n/2^k$ lists of length 2^k into $n/2^{k+1}$ lists of length 2^{k+1} . A little math suggests that this once again takes approximately n steps.

So far, so good. Now, how many times do we call `merge-pairs`. We go from lists of length 1, to lists of length 2, to lists of length 4, to lists of length 8, ..., to lists of length $n/4$, to lists of length $n/2$, to one list of length n . How many times did we call `merge-pairs`? The number of times we need to multiply 2 by itself to get n . As I've noted before, the formal name for that value is $\log_2 n$.

To conclude, merge sort repeats a step of n steps $\log_2 n$ times. Hence, it takes approximately $n \log_2 n$ steps.

Here's a chart that will help you compare various running times.

n	$\log_2 n$	n^2	$n^2/4$	$n \log_2 n$
10	3.3	100	25	33
20	4.3	400	100	86
30	4.9	900	225	147
40	5.3	1,600	400	212
100	6.6	10,000	2,500	660
500	9.0	250,000	62,500	4,483
1000	10.0	1,000,000	250,000	10,000

As you can see, although the two sorting algorithms start out taking approximately the same time, as the length of the list grows, the relative cost of using insertion sort becomes a bigger and bigger ratio of the cost of using merge sort.

In the laboratory, you'll have an opportunity to analyze experimentally how many steps each algorithm uses.

Documenting Merge Sort

You may have noted that we have not yet written the documentation for merge sort. Why not? Because it's basically the same as the documentation for any other sorting routine. Here it is, for completeness.

```
;;; 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 and reflexive 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 are permutations of each other.  
;;; Does not affect stuff.  
;;; sorted may share cons cells with stuff.
```

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.