Quicksort

Summary: In a recent reading, you explored *merge sort*, a comparatively efficient algorithm for sorting lists or vectors. In this reading, we consider one of the more interesting sorting algorithms, Quicksort.

Contents:

Alternative Strategies for Dividing Lists

As you may recall, the two key ideas in merge sort are: (1) use the technique known as of *divide and conquer* to divide the list into two halves (and then sort the two halves); (2) merge the halves back together. As we saw in both the reading and the corresponding lab, we can divide the list in almost any way.

Are there better sorting algorithms than merge sort? If our primary activity is to compare values, we cannot do better than some constant times $n\log_2 n$ steps in the sorting algorithm. However, that hasn't stopped computer scientists from exploring alternatives to merge sort. (In the next course, you'll learn some reasons that we might want such alternatives.)

One way to develop an alternative to merge sort is to split the values in the list in a more sensible way. For example, instead of splitting into "about half the elements" and "the remaining elements", we might choose the to divide into "the smaller elements" and "the larger elements".

Why would this strategy be better? Well, if we know that every small element precedes every large element, then we can significantly simplify the merge step: We just need to append the two sorted lists together, with the equal elements in the middle.

```
(define new-sort
  (lambda (lst precedes?)
    (if (or (null? lst) (null? (cdr lst)))
        lst
        (let ((smaller (small-elements lst precedes?))
              (same (equal-elements lst precedes?))
              (larger (large-elements lst precedes?)))
          (append (new-sort smaller) same (new-sort larger))))))
```

You'll note that we used `precedes?` rather than the `may-precede?` that we used in previous sorting algorithms. That's because we really want to segment out the strictly larger and strictly smaller values. (Other variants of this sorting algorithm might include the equal elements in one side or the others. However, such versions require consideration of some subtleties that we'd like to avoid.)

Identifying Small and Large Elements

It sounds like a great idea, doesn't it? Instead of `split` and `merge`, we can sort by writing `small-elements`, `equal elements`, and `large-elements`. So, how do we write those procedures? A statistician might tell us that "the small elements are all the elements less than the median" and that "the large elements are the elements greater than the median". That's a pretty good starting definition. Now, how do we find the median? Usually, we sort the values and look at the middle position. Whoops. If we need the median to sort, and we need to sort to get the median, we're stuck in an overly recursive situation.

So, what do we do? A computer scientist named C. A. R. Hoare had an interesting suggestion: *Randomly pick some element of the list and use that as a simulated median.* That is, anything smaller than that element is "small" and anything larger than that element is "large". Because it's not the median, we need another name for that element. Traditionally, we call it the *pivot*. Is using a randomly-selected pivot a good strategy? You need more statistics than most of us know to prove formally that it works well. However, practice suggests that it works very well.

We can now write `small-elements`, `equal-elements`, `large-elements` by including the pivot.

```
(define small-elements
  (lambda (lst precedes? pivot)
    (cond
      ((null? lst) null)
      ((precedes? (car lst) pivot)
       (cons (car lst) (small-elements (cdr lst) precedes? pivot)))
      (else
       (small-elements (cdr lst) precedes? pivot)))))

(define large-elements
  (lambda (lst precedes? pivot)
    (cond
      ((null? lst) null)
      ((precedes? pivot (car lst))
       (cons (car lst) (large-elements (cdr lst) precedes? pivot)))
      (else
       (large-elements (cdr lst) precedes? pivot)))))

(define equal-elements
  (lambda (lst precedes? pivot)
    (cond
      ((null? lst) null)
      ((and (not (precedes? pivot (car lst)))
            (not (precedes? (car lst) pivot)))
       (cons (car lst) (equal-elements (cdr lst) precedes? pivot)))
     (else
      (equal-elements (cdr lst) precedes? pivot)))))
```

You may note that `equal-elements` does not use `equal?` to compare the pivot to the car. Why not? Because our ordering may be more subtle. For example, we may be comparing people by height (one of the many values we store for each person), and two different people can still be equal in terms of height.

Once we've defined these three procedures, we then have to update our main sorting algorithm to find the pivot and to use it in identifying small, equal, and large elements. Since we use the sublists only once, we won't even bother naming them. We'll call the new algorithm Quicksort, since that's what Hoare called it.

```
(define Quicksort
  (lambda (lst precedes?)
    (if (or (null? lst) (null? (cdr lst)))
        lst
        (let ((pivot (random-element lst)))
          (append (Quicksort (small-elements lst precedes? pivot)
                             precedes?)
                  (equal-elements lst precedes? pivot)
                  (Quicksort (large-elements lst precedes? pivot)
                             precedes?))))))
```

Selecting a Pivot

How do we select a random element from the list? We've done so many times before that the code should be self explanatory.

```
(define random-element
  (lambda (lst)
    (list-ref lst (random (length lst)))))
```

Refactoring

Are we done? In one sense, yes, we have a working sorting procedure. However, good design practice suggests that we look for ways to simplify or otherwise clean up our code. What are the main principles? (1) Don't name anything you don't need to name. (2) Don't duplicate code.

The only thing we've named is the pivot. We've used it three times, which argues for naming it. More importantly, since the pivot is a randomly chosen value, our code will not work the same (nor will it work correctly) if we substitute `(random-element lst)` for each of the instances of `pivot`. Hence, the naming is a good strategy.

Do we have any duplicated code? Yes, `small-values`, `equal-values`, and `large-values` are very similar. Each scans a list of values and selects those that meet a particular predicate (in the first case, those less than the pivot, in the second, those equal to the pivot, in the third, those greater than to the pivot). Hence, we might want to write a `select` procedure that extracts the similar code.

```
;;; Procedure:
;;;   select
;;; Parameters:
;;;   pred?, a unary predicate
;;;   lst, a list
;;; Purpose:
```

```
;;;   Create a list of all values in lst for which pred? holds.
;;; Produces:
;;;   selected, a list
;;; Preconditions:
;;;   pred? can be applied to each element of lst.
;;; Postconditions:
;;;   Every element in selected is in lst.
;;;   pred? holds for every element of selected.
;;;   If there's a value in lst for which pred? holds, then the value is in
;;;     selected.
(define select
  (lambda (pred? lst)
    (cond
      ((null? lst) null)
      ((pred? (car lst))
       (cons (car lst) (select pred? (cdr lst))))
      (else (select pred? (cdr lst))))))
```

Now, we can write Quicksort without relying on the helpers `small-elements` and `large-elements`.

```
(define Quicksort
  (lambda (lst precedes?)
    (if (or (null? lst) (null? (cdr lst)))
        lst
        (let ((pivot (random-element lst)))
          (append (Quicksort
                   (select (lambda (val) (precedes? val pivot)) lst)
                   precedes?)
                  (select (lambda (val) (and (not (precedes? val pivot))
                                             (not (precedes? pivot val))))
                          lst)
                  (Quicksort
                   (select (lambda (val) (precedes? pivot val)) lst)
                   precedes?))))))
```

Can we make this even shorter? Well, the selection of large elements looks a lot like a use of `left-section` and the selection of small elements is a lot like a use of `right-section`. The selection of equal elements we may just have to leave in its more complex form.

Now we can write something even more concise.

```
(define Quicksort
  (lambda (lst precedes?)
    (if (or (null? lst) (null? (cdr lst)))
        lst
        (let ((pivot (random-element lst)))
          (append (Quicksort (select (r-s precedes? pivot) lst) precedes?)
                  (select (lambda (val) (and (not (precedes? val pivot))
                                             (not (precedes? pivot val))))
                          lst)
                  (Quicksort (select (l-s precedes? pivot) lst) precedes?))))))
```

An Alternate Strategy for Building the Lists of Small, Equal, and Large Values

Some designers might focus not on the duplication of code between `small-elements`, `equal-elements`, and `large-elements` and instead focus on the issue that all we're really doing is splitting `lst` into three lists. They might suggest that instead of writing `select`, we write a `partition` procedure that breaks a list into three parts.

```
;;; Procedure:
;;;   partition
;;; Parameters:
;;;   lst, a list
;;;   pivot, a value
;;;   precedes?, a binary predicate
;;; Purpose:
;;;   Partition lst into three lists,
;;;    one for which (precedes? val pivot) holds,
;;;    one for which (precedes? pivot val) holds, and
;;;    one for which neither holds.
;;; Produces:
;;;   (smaller-elements equal-elements larger-elements), A two element list
;;; Preconditions:
;;;   precedes? can be applied to pivot and any value of lst.
;;; Postconditions:
;;;   (append smaller-elements equal-elements larger-elements)
;;;     is a permutation of lst.
;;;   (precedes? (list-ref smaller-elements i) pivot)
;;;     holds for every i, 0 < i < (length smaller-elements).
;;;   (precedes? pivot (list-ref larger-elements j))
;;;     holds for every j, 0 < j < (length larger-elements).
;;;   Neither (precedes? (list-ref equal-elements k) pivot)
;;;     nor (precedes? pivot (list-ref equal-elements k))
;;;     holds for eery k, 0 < k < (length equal-elements)
(define partition
  (lambda (lst pivot precedes?)
    (letrec ((kernel
               (lambda (remaining smaller-elements equal-elements larger-elements)
                 (cond
                   ((null? remaining)
                    (list smaller-elements equal-elements larger-elements))
                   ((precedes? (car remaining) pivot)
                    (kernel (cdr remaining)
                            (cons (car remaining) smaller-elements)
                            equal-elements
                            larger-elements))
                   ((precedes? pivot (car remaining))
                    (kernel (cdr remaining)
                            smaller-elements
                            equal-elements
                            (cons (car remaining) larger-elements)))
                   (else
                    (kernel (cdr remaining)
                            smaller-elements
                            (cons (car remaining) equal-elements)
                            larger-elements))))))
      (kernel lst null null null))))
```

Here’s yet another version of Quicksort that uses this procedure.

```
(define Quicksort
  (lambda (lst precedes?)
    (display lst) (newline)
    (if (or (null? lst) (null? (cdr lst)))
        lst
        (let* ((pivot (random-element lst))
               (parts (partition lst pivot precedes?)))
          (append (Quicksort (car parts) precedes?)
                  (cadr parts)
                  (Quicksort (caddr parts) precedes?))))))
```

Which Quicksort is Best?

You’ve now seen four versions of Quicksort. Which one is best? The last one is probably the most efficient, since it only scans the list once to identify the small elements and large elements. The other three scan the list twice.

Different readers find different versions clearer or less clear. You’ll need to decide which you like the most from that perspective (and you might even want to think about how you’d express the criteria by which you make your decision).

Copyright © 2006 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.