

QUICKSORT IN SCHEME

The version of quicksort below has some commented code, but the uncommented code constitutes a complete running version of quicksort. The commented code is what one might write as a first pass -- whenever an object is needed, we just write down the SCHEME expression that will produce it. Then, we observe that certain computations are being performed redundantly. Specifically, the partitioning of the list into two lists of elements smaller than, and greater than or equal to, the splitting element. So we use the "let*" function to provide some local variables in which we can store the results of intermediate computation. These results can then be accessed without repeating the computation.

The function "partition" could be developed in exactly the same way. However, it is written instead to illustrate another technique for storing intermediate results. Instead of extra variables defined through `let*`, extra formal parameters are used. The last two parameters are the two lists that are (eventually) returned. In effect, we use extra formal parameters to store partial results as the recursion proceeds. In the base case of the recursion, there are no more elements in `l` to be added to the lists of the partition, so we just return a list whose elements are `less & gte`. This technique will be useful when we study Prolog, because in that language there are no constructs comparable to `let*`. So it is very common to use parameters as "accumulators" for results being computed.

```
; this function uses the random number generator to create  
; a list of integers of length in the range 1 - 100000.
```

```
(define makelist (lambda (n)  
  (set! *random-state* (make-random-state #t))  
  (mkl n) ))
```

```
(define mkl (lambda (n)  
  (cond ((< n 2) (list (random 100000)))  
        (#t (cons (random 100000)  
                   (mkl (- n 1)) ) ) ) ) )
```

```
(define compares 0) ; global counter
```

```
(define qsort (lambda (L)  
  (let ((newL (quicksort L)))  
    (write compares) (write-string " comparisons used")  
    (set! compares 0) newL ) ) )
```

```
(define quicksort (lambda (L) ; this is quicksort  
  (cond  
    ((null? L) L)  
    ((null? (cdr L)) L)  
    ; (#t (append  
    ;   (quicksort  
    ;     (car (partition (car L) (cdr L) () ())) )  
    ;   (cons (car L)  
    ;     (quicksort  
    ;       (cadr (partition (car L) (cdr L) () ()))  
    ; ) ) ) ) ) ) ) ) )
```

```

(define quicksort (lambda (L) this is quicksort
  (cond
    ((null? L) L)
    ((null? (cdr L)) L)
;   (#t (append
;     (quicksort
;       (car (partition (car L) (cdr L) () ())) )
;     (cons (car L)
;       (quicksort
;         (cadr (partition (car L) (cdr L) () ()))
;       ) ) ) ) ) ) )
;   (#t (let*
;     ((par (partition (car L) (cdr L) () ()))
;      (less (quicksort (car par)))
;      (gte (quicksort (cadr par))) )
;     (append less (cons (car L) gte)) ) ) ) ) )

```

```

(define partition (lambda (sp l less gte)
  (cond
    ((null? l) (list less gte))
    (#t (set! compares (+ 1 compares))
      (cond
        ((< (car l) sp)
          (partition sp (cdr l) (cons (car l) less) gte) )
        (#t
          (partition sp (cdr l) less (cons (car l) gte))
        )
      ) ) ) ) )

```

```

; this function uses the random number generator to create
; a list of integers of length in the range 1 - 100000.
;
(define makelist (lambda (n)
  (set! *random-state* (make-random-state #t))
  (mkl n) ))

(define mkl (lambda (n)
  (cond ((< n 2) (list (random 100000)))
        (#t (cons (random 100000)
                   (mkl (- n 1)) ) ) ) ))

(define compares 0) ; global counter

; invokes the insertion sort function and prints
; #compares. then resets the global counter to zero.
;
(define isort (lambda (L)
  (let ((newL (isrt L)))
    (write compares) (write-string " comparisons used")
    (set! compares 0) newL ) ))

```

```
; this is a simple insertion sort
```

```
;
```

```
(define isrt (lambda (L)
  (cond ((null? L) ())
        ((null? (cdr L)) L)
        (#t (place (car L)
                    (isrt (cdr L)) )) ) ) )
```

```
; place assumes L is a sorted list and returns a new list
```

```
; that has element n and all the elements of L.
```

```
;
```

```
(define place (lambda (n L)
  (cond
    ((null? L) (list n))
    (#t (set! compares (+ 1 compares))
         (cond
           ((<= n (car L))
            (cons n L) )
           (#t (cons (car L)
                     (place n (cdr L)) )) ) ) ) ) )
```

```
; invokes the mergesort function and prints #comparisons.  
; then resets the global counter to zero.  
;
```

```
(define msort (lambda (L)  
  (let ((newL (msrt L)))  
    (write compares) (write-string " comparisons used")  
    (set! compares 0) newL ) ))
```

```
; mergesort -- guaranteed to have  $n \log n$  complexity  
;
```

```
(define msrt (lambda (L)  
  (cond ((null? L) ())  
        ((null? (cdr L)) L)  
        (#t (let ((pofl (mk2 L)))  
              (merge (msrt (car pofl))  
                    (msrt (cadr pofl)) ) ) ) ) ))
```



```
1 ]=> (define lst (make-list 500))
```

```
;Value: lst
```

```
1 ]=> lst
```

```
;Value 1: (67424 99470 38701 663 37175 .....
```

```
1 ]=> (qsort lst)
```

```
4699 comparisons used
```

```
;Value 2: (52 256 590 658 663 675 806 .....
```

```
1 ]=> (isort lst)
```

```
62398 comparisons used
```

```
;Value 3: (52 256 590 658 663 675 806 .....
```

```
1 ]=> (define slst (qsort lst))
```

```
4699 comparisons used
```

```
;Value: slst
```

```
1 ]=> (qsort slst)
```

```
124637 comparisons used
```

```
;Value 4: (52 256 590 658 663 675 806 .....
```

```
1 ]=> (isort slst)
```

```
499 comparisons used
```

```
;Value 5: (52 256 590 658 663 675 806 .....
```

1]=> (msort lst)

3828 comparisons used

;Value 6: (52 256 590 658 663 675 806

1]=> (msort slst)

3989 comparisons used

;Value 7: (52 256 590 658 663 675 806

1]=> (btsort lst)

4750 comparisons used

;Value 8: (52 256 590 658 663 675 806

1]=> (btsort slst)

124426 comparisons used

;Value 9: (52 256 590 658 663 675 806