# Some Recursive List Processing Algorithms in Lisp

Michael A. Covington



## 1 Key idea

A list is a recursive data structure. If not empty, it consists of one element followed by another list.

Thus, there are many recursive algorithms that process a list by doing something to the first element and then recursively processing the REST (the CDR) of the original list. This is called "CDRing down" the list.

Such an algorithm must always check whether the list is empty, and if so, terminate the recursion.

*Note:* This is not the only way to build recursive algorithms in Lisp. Instead of CDRing down a list, some algorithms decrement or increment a number or control their recursion in some other way.

## 2 Searching for an element

Suppose you want to know whether X is an element of list L. There are three possibilities:

- If L is empty, X is not an element of it.
- If (FIRST L) equals X, you've found it.
- Otherwise, X is in L if and only if X is in (REST L).

The last of these is, of course, recursive.

Here's the algorithm, in Lisp:

```
(defun mem (E L)
  (if (null L)
      nil
      (if (equal (first L) E)
          T
          (mem E (rest L))
      )
  )
)
```

Then:

`(mem 'A '(B A G))` ⇒ `T`

`(mem 'Z '(B A G))` ⇒ `NIL`

`(mem 'A '())` ⇒ `NIL`

`(mem 'A '(B (A A) G))` ⇒ `NIL`

In the last of these, (A A) is an element of the list, but A is not.

(There is a somewhat similar built-in function named `member`, which we're not using.)

**Exercise 2.1** *What happens if X occurs more than once in L?*

It turns out to be more useful if, instead of returning T when the E is found, we return the sublist of L that begins with E.

That is, when `(first L)` is what we are looking for, we should return L rather than T. This is always non-nil, so we can still use `mem` as a predicate, but the value of L is sometimes useful to the calling function.

**Exercise 2.2** *Modify* `mem` *so that it returns L instead of T. It should then work this way:*

```
(mem 'A '(B A G)) ⇒ (A G)
(mem 'Z '(B A G)) ⇒ NIL
(mem 'A '(Z E B R A S A N D C O W S))
                  ⇒ (A S A N D C O W S)
```

**Exercise 2.3** *Suppose L is a list, and you have modified* `mem` *as indicated in the previous exercise. Under what conditions will*

```
(mem 'A (rest (mem 'A L)))
```

*return a non-nil value? That is, what does the value of L have to be?*

**Exercise 2.4** *Does it take more memory for* `mem` *to return a list rather than just returning T or NIL? Explain.*

# 3 Copying a list

Here is a useless but interesting recursive algorithm to copy a list:

```
(defun copylist (L)
   (if (null L)
       nil
       (cons (first L)
             (copylist (rest L))
       )
   )
)
```

That is: To copy a list, CONS its FIRST with a copy of its REST. That is, to copy (A B C), the computer would end up executing something analogous to

```
(cons 'A (cons 'B (cons 'C NIL)))
```

except of course there are no quotes because the symbols A, B, and C are the results of computation.

**Exercise 3.1** *Get* `copylist` *working and evaluate the following expressions:*

```
(copylist '(A B C))
```

```
(equal '(A B C) (copylist '(A B C)))
```

[*testing whether they contain the same data in the same arrangement*]

```
(eq '(A B C) (copylist '(A B C)))
```

[*testing whether they are stored in the same memory location*]

**Exercise 3.2** *Consider this version of* `copylist`*:*

```
(defun copylist2 (L)
   (if L
       (cons (first L)
             (copylist2 (rest L))
       )
   )
)
```

*Is it the same algorithm? What has been changed?*

# 4 Appending lists

This algorithm is more useful if we don't actually copy the whole list unchanged. Suppose that we put another list, such as '(D E F), in place of the final NIL. Then the computer would execute the equivalent of

```
(cons 'A (cons 'B (cons 'C '(D E F))))
```

and give us (A B C D E F). We've found a way to APPEND (concatenate) lists.

Here's the list-appending algorithm implemented as a function. It has two arguments. The first is the list that it will be CDRing down, and the second holds the list that will go at the end.

```
(defun app (L1 L2)
  (if (null L1)
      L2
      (cons (first L1)
            (app (rest L1) L2)
      )
  )
)
```

(There is a similar built-in function named `append`, which we're not using.)

**Exercise 4.1** *Get* `app` *working and try these evaluations:*

```
(app '(a b c) '(d e f))
(app '((a b c)) '((d e f)))
```

*What's different about the second one?*

**Exercise 4.2** *Which of the following takes more time and memory? Why?*

```
(app '(a b) '(c d e f g h))
(app '(a b c d e f) '(g h))
```

**Exercise 4.3** *Are there any restrictions on the types of the elements of the lists that can be joined by* `app`*?*

**Exercise 4.4** *Under what condition does* `app` *return nil?*

# 5 Making an altered copy of a list

Algorithms like `copylist` are useful when you want to make a list that is similar to the first one but has undergone certain changes. Here is a simple example of how to copy a list of numbers, adding 1 all the numbers as they are copied:

```
(defun add-one-list (L)
   (if (null L)
       nil
       (cons (+ 1 (first L))
             (add-one-list (rest L))
       )
   )
)
```

Given a list of numbers, `add-one-list` does things like this:

```
(add-one-list '(20 30 90))
                          ⇒ (21 31 91)
```

**Exercise 5.1** *Get* `add-one-list` *working and demonstrate that it works as advertised.*

# 6 Extracting elements

Another variation on `copylist` is to copy some of the elements but not all of them. Here's an example that works on lists of numbers. (Its funny-looking name `list<` is a valid symbol in Lisp.)

```
(defun list< (N L)
  ; Makes a copy of list L
  ; containing only elements
  ; that are < N.
  ; L must be a list of numbers.
  (if (null L)
      nil
      (if (< (first L) N)
          (cons (first L)
                (list< N (rest L))
          )
          (list< N (rest L))
      )
   )
)
```

For example:

```
(list< 5 '(4 9 4 6 3 3)) ⇒ (4 4 3 3)
```

**Exercise 6.1** *Define* `list>=`*, which is just like* `list<` *except that it extracts the elements that are* $\geq N$*.*

*(Hint: This is as easy as it looks. You even have a built-in function* `>=` *that does what you need.)*

**Exercise 6.2** *Define* `skip`*, which omits the elements that are* `equal` *to* $N$ *and copies all the others. In this case neither* $N$ *nor the list elements need be numbers. Example:*

```
(skip 'A '(B A D L A N D S))
                          ⇒ (B D L N D S)
```

**Exercise 6.3** *In the definition of* `list<` *above, the expression*

`(list< N (rest L))`

*is written twice. Does this indicate wasteful computation? Explain.*

## 7 Quicksort

Now that we have `app`, `list<`, and `list>=`, we have everything necessary to implement Quicksort, an algorithm invented by C. A. R. Hoare in 1961 and originally implemented in Algol (the first programming language that allowed recursion).

Quicksort is an algorithm for putting elements in order; for example, it turns the list (5 4 9 4 6 3 3) into (3 3 4 4 5 6 9). Quicksort is efficient; it can sort an $n$-element list in time proportional to $n \log n$, whereas selection sort and similar algorithms require time proportional to $n^2$.

What's more, unlike selection sort and its kin, Quicksort does not require you to swap or move elements of arrays. Thus, Quicksort can be done on lists without manipulating pointers.

The key idea of Quicksort is that sorting is recursive. Take the list that you're going to sort, say

(5 4 9 4 6 3 3)

and pick out the first element. Now you have:

5 (4 9 4 6 3 3)

Now divide the list (4 9 4 6 3 3) into two lists, one containing elements that should come before 5 and one containing elements that should come after it:

(4 4 3 3) 5 (9 6)

Now *recursively sort the two shorter lists*:

(3 3 4 4) 5 (6 9)

Finally, put everything back together in order, and you have:

(3 3 4 4 5 6 9)

The recursion terminates when a list is empty, because you can sort an empty list without doing anything to it.

You've probably just realized that `list<` and `list>=` are crucial steps in Quicksort. In fact, if you have already defined them, the Quicksort algorithm is simply:

```
(defun qsort (L)
  (if (null L)
      nil
      (append
         (qsort (list<  (first L) (rest L)) )
         (cons (first L) nil)
         (qsort (list>= (first L) (rest L)) )
      )
  )
)
```

Here we're using the built-in function `append`, which can append 3 lists in one step, rather than our own `app`, which only appends two.

**Exercise 7.1** *Get Quicksort working and try it on lists of numbers.*

**Exercise 7.2** *In Quicksort, why do we use*

`(cons (first L) nil)`

*instead of just* `L`*?*

— END —