

## CM20167 Topic 6: Some sorting algorithms

Guy McCusker

1W2.1

### Functional programming is clean and clear

This lecture attempts to support the claim that:

Functional programming is a clean, clear way of expressing algorithms.

To illustrate this, we'll write code for three different sorting algorithms in this lecture:

- ▶ Quicksort
- ▶ Insertion sort
- ▶ Merge sort

### Quicksort

Quicksort is an algorithm for sorting a list of data into order, developed by Sir C. A. R. Hoare in 1960.

It has a special place among sorting algorithms because its performance, on average, is the best possible, and in practice it is the fastest.

It has a special place in functional programming courses because it's very easy to write!

## The qsort algorithm

The algorithm is as follows:

- ▶ Pick an element, called a pivot, from the list.
- ▶ Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
- ▶ Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

(Description from Wikipedia:

<http://en.wikipedia.org/wiki/Quicksort>.)

## Our qsort

In our implementation, we will take as a pivot the car of the list we're sorting.

"Reorder the list" for us will mean "split the list into three parts: those elements less than the pivot; the pivot itself; and those elements greater than the pivot."

This will allow us to perform the recursive calls, and then we'll concatenate the results together.

## Splitting the list

To take the collection of elements of a list which are smaller than the car, we filter the list's cdr:

```
(filter something (cdr l))
```

What should the filter be? It needs to be a predicate which returns `t` if the element it finds is smaller than `car l`:

```
(lambda (a) (< a (car l)))
```

## Splitting the list

So the list of elements smaller than the car is:

```
(filter (lambda (a) (< a (car l))) (cdr l))
```

and the list of elements not smaller than the car is:

```
(filter (lambda (a) (not (< a (car l)))) (cdr l))
```

## Recursively sorting the smaller parts

To sort the whole list, we recursively sort those smaller parts and then concatenate the whole lot together:

```
(concatenate
  (qsort (filter (lambda (a) (< a (car l))) (cdr l)))
  (list (car l))
  (qsort (filter (lambda (a) (not (< a (car l)))) (cdr l)))
)
```

## The qsort function

```
(defun qsort (l)
  (if (null l)
      nil
      (concatenate
        (qsort (filter (lambda (a) (< a (car l))) (cdr l)))
        (list (car l))
        (qsort (filter (lambda (a) (not (< a (car l)))) (cdr l)))
      )
  )
)
```

## Tidy up with let

It's (arguably) slightly cleaner to tidy this function up using `let` to name the two "halves" of the list:

```
(defun qsort (l)
  (if (null l)
      nil
      (let ((l1 (filter (lambda (a) (< a (car l))) (cdr l)))
            (l2 (filter (lambda (a) (not (< a (car l))) (cdr l))))
        (concatenate (qsort l1) (list (car l)) (qsort l2))
        )
      )
  )
)
```

## Insertion sort

Insertion sort is a very different sorting algorithm.

It processes each element of the input list in turn, and maintains a partial result consisting of a list of the elements we've seen already, sorted into order.

To process the next element, we simply insert it into the partial result list, in the appropriate place.

## Inserting

First let's write a function to insert an element into an ordered list without disrupting the order.

We'll find the right place by a simple linear search.

To insert element `a` into list `l`:

- ▶ If `l` is empty, just return `(a)`
- ▶ If `a` comes before `(car l)`, put `a` onto the front of `l`.
- ▶ Otherwise, make `(car l)` the first element and insert `a` into `(cdr l)`.

## Insert

```
(defun insert (a l)
  (cond ((null l) (list a))
        ((< a (car l)) (cons a l))
        (t (cons (car l) (insert a (cdr l)))))
  )
)
```

To insert element *a* into list *l*:

- ▶ If *l* is empty, just return (*a*)
- ▶ If *a* comes before (*car l*), put *a* onto the front of *l*.
- ▶ Otherwise, make (*car l*) the first element and insert *a* into (*cdr l*).

## Insert

```
(defun insert (a l)
  (cond ((null l) (list a))
        ((< a (car l)) (cons a l))
        (t (cons (car l) (insert a (cdr l)))))
  )
)
```

To insert element *a* into list *l*:

- ▶ If *l* is empty, just return (*a*)
- ▶ If *a* comes before (*car l*), put *a* onto the front of *l*.
- ▶ Otherwise, make (*car l*) the first element and insert *a* into (*cdr l*).

## Insert

```
(defun insert (a l)
  (cond ((null l) (list a))
        ((< a (car l)) (cons a l))
        (t (cons (car l) (insert a (cdr l)))))
  )
)
```

To insert element *a* into list *l*:

- ▶ If *l* is empty, just return (*a*)
- ▶ If *a* comes before (*car l*), put *a* onto the front of *l*.
- ▶ Otherwise, make (*car l*) the first element and insert *a* into (*cdr l*).

## Computing the sorted list

Now if we want to sort the list  $(a_1 a_2 a_3 \dots a_n)$ , we need to compute

```
(insert a1 (insert a2 (insert a3 ... (insert an ())))))
```

or to put it another way

```
(defun isort (l)
  (foldr insert () l)
)
```

## Or using foldl...

We could write it with `foldl` too:

```
(defun isortl (l)
  (foldl (flip insert) () l)
)
```

where the `flip` function swaps the order of arguments:

```
(defun flip (f)
  (lambda (a b) (f b a))
)
```

## Mergesort

Mergesort is a very cute recursive algorithm, and it's theoretically optimal in terms of speed.

It works like this:

- ▶ Split the list to be sorted into two smaller lists of about half the size
- ▶ Recursively sort those sublists
- ▶ Merge the two sorted sublists together, maintaining the order.

So there are two key operations we need to implement:

- ▶ splitting a list into two halves
- ▶ merging two ordered lists into one.

## Merging

Suppose we have two sorted lists `l1` and `l2`, and we wish to merge them into one sorted list.

What we do is:

- ▶ if `l1` is empty, return `l2`.
- ▶ if `l2` is empty, return `l1`.
- ▶ if both are non-empty, compare their first elements.
  - ▶ the smaller of the two first elements is the first element of the result list
  - ▶ compute the rest of the result list by a recursive call.

## Merging

```
(defun merge (l1 l2)
  (cond ((null l1) l2)
        ((null l2) l1)
        (t (if (< (car l1) (car l2))
                (cons (car l1) (merge (cdr l1) l2))
                (cons (car l2) (merge l1 (cdr l2))))
         )
  )
)
```

## A different-looking recursive call

The recursive call here looks different to those we've seen before: there are two possible recursive calls,

- ▶ `(merge (cdr l1) l2)`
- ▶ `(merge l1 (cdr l2))`

Note that we're taking the `cdr` of *either* `l1` or `l2`, not both.

That means we're not simply using recursion on one of the lists, but on both together.

It's not a problem, of course; but it is something a bit different.

## Splitting the list

Splitting the input list in two is straightforward. We define two auxiliary functions:

```
(defun take (n l) ;; returns the first n elements of l
  (cond ((zerop n) nil)
        ((null l) nil)
        (t (cons (car l) (take (- n 1) (cdr l))))
  )

(defun drop (n l) ;; returns l without its first n elements
  (cond ((zerop n) l)
        ((null l) nil)
        (t (drop (- n 1) (cdr l)))
  )
)
```

## Split, sort, merge, done

Now mergesort is easy to write:

```
(defun mergesort (l)
  (if (< (length l) 2) ;; zero or one elements
      l
      ;; at least two elements, so split, sort and merge
      (let ((first (take (/ (length l) 2) l))
            (second (drop (/ (length l) 2) l)))
          (merge (mergesort first) (mergesort second)))
  )
)
```

## Abstract!

These sorting algorithms were easy to write in Lisp.

However, they only work for sorting lists whose elements can be compared using `<`. This works for numbers and strings, but probably not much else.

It would be better to allow an arbitrary comparison function, so that we can sort different kinds of list.

Suppose we have a comparison function `comp` such that

$$(\text{comp } a \text{ } b)$$

returns `t` if we should put `a` before `b` in the sorted list.

We'd like to be able to specify the comparison operator when we call `mergesort`, with something like

$$(\text{mergesort } \text{mylist } \text{comp})$$

## A more general merge

The first thing to do is to update merge, which is where the comparisons take place.

First we replace < with comp.

Then we make comp an argument to the function

```
(defun merge (l1 l2 comp)
  (cond ((null l1) l2)
        ((null l2) l1)
        (t (if (comp (car l1) (car l2))
                (cons (car l1) (merge (cdr l1) l2 comp))
                (cons (car l2) (merge l1 (cdr l2) comp))
            )
        )
  )
)
```

## Updating mergesort

Now that merge takes an extra argument, we need to change mergesort in the same way.

We pass it an extra argument, which gets passed on to all recursive calls, and to all the merges.

```
(defun mergesort (l comp)
  (if (< (length l) 2) ;; zero or one elements
      l
      ;; at least two elements, so split, sort and merge
      (let ((first (take (/ (length l) 2) l))
            (second (drop (/ (length l) 2) l)))
          (merge (mergesort first comp)
                  (mergesort second comp) comp)
        )
      )
)
```

## Try it out

As a first test, let's see if we can sort into both ascending and descending order:

```
user> (mergesort '(1 4 3 2 5 6 8 7 9) <)
(1 2 3 4 5 6 7 8 9)
user> (mergesort '(1 4 3 2 5 6 8 7 9) >)
(9 8 7 6 5 4 3 2 1)
```

## More interesting sorts

Suppose I've got a list of triples of the form

```
(123456 "BLOGGS" "F")
```

containing the candidate numbers, surnames and initials of all students on CM20167.

(Which I have.)

I can sort it in three different ways:

- ▶ By candidate number:  
`(mergesort studentlist (lambda (a b) (< (car a) (car b))))`
- ▶ By surname:  
`(mergesort studentlist (lambda (a b) (< (cadr a) (cadr b))))`
- ▶ By initial:  
`(mergesort studentlist (lambda (a b) (< (caddr a) (caddr b))))`

## Generalizing quicksort

It's very easy to generalize the quicksort function too: add the extra argument, use `comp` instead of `<`, and remember to pass `comp` on to the recursive calls.

```
(defun qsort (l comp)
  (if (null l)
      nil
      (let ((l1 (filter (lambda (a) (comp a (car l))) (cdr l)))
            (l2 (filter (lambda (a) (not (comp a (car l))) (cdr l))))
            (concatenate
              (qsort l1 comp)
              (list (car l))
              (qsort l2 comp)))
        )
      )
  )
```

## Generalizing insertion sort

It's a little trickier to generalize our insertion sort in the same way, because we used `foldr`.

```
(defun insertionsort (l)
  (foldr insert () l)
)
```

Since `foldr` is handling our recursion, it can't readily be adapted to pass on the extra `comp` argument to the recursive calls.

What we need to do is pass `comp` to the *insertion* function:

```
(defun isort-gen (l comp)
  (foldr (insert-gen comp) () l)
)
```

## Generalized insert

The generalized `insert` function receives an argument `comp` to use as its comparison.

`(insert-gen comp)` should itself be a *function*: the function that does the appropriate insertion, using `comp`.

That function looks like this:

```
(lambda (a l)
  (cond ((null l) (list a))
        ((comp a (car l)) (cons a l))
        (t (cons (car l) ((insert-gen comp) a (cdr l)))))
  )
)
```

## Putting it all together

```
(defun insert-gen (comp)
  (lambda (a l)
    (cond ((null l) (list a))
          ((comp a (car l)) (cons a l))
          (t (cons (car l) ((insert-gen comp) a (cdr l)))))
    )
  )

(defun isort-gen (l comp)
  (foldr (insert-gen comp) () l)
  )
```

## Sneaky comparators

Now that we can pass any old function as a comparison operator, we can do some interesting things.

For instance, we could pass a function which not only performs the comparison, but also prints out a tick mark, showing that a comparison has happened. That way, we can compare the performance of the various sorting algorithms.

### Warning

We are leaving the realm of pure functional programming now, and getting into *side effects*.

## Side effects

A function which not only returns a value but also causes something else to happen:

- ▶ something gets printed
- ▶ some other input/output takes place
- ▶ the value of a variable is changed

is said to have a *side-effect*.

So far, the only side-effect we've seen in Lisp was the use of `setq` to put a value into a variable.

A useful side-effecting expression is

```
(print something)
```

which evaluates `something`, prints that value to the output, and then returns the value as an answer too.

## Sequencing

When using side-effects, it's common to want to perform several side-effecting computations, one after the other.

Lisp lets you do this with `progn`:

```
(progn exp1 exp2 ... expn)
```

evaluates the expressions `exp1`, ..., `expn` sequentially, and returns the value of the last one. So

```
(progn (print ".") exp)
```

has the same value as `exp`, but prints a dot along the way.

## Instrumented comparison

We can now define a comparison function which behaves the same as `<` but prints a dot each time it is called:

```
(defun icomp (a b)
  (progn (print ".") (< a b))
)
```

Using this in place of `<` gives us an idea of how many comparisons our sorting algorithms are performing.

## Trying it

```
user>(isort-gen '(1 5 2 4 3) icomp)
.
.
.
.
.
(1 2 3 4 5)
```

## Comparing sorts

It turns out that when sorting the list (1 5 2 4 3),

- ▶ quicksort takes 20 comparisons
- ▶ mergesort takes 7 comparisons
- ▶ insertion sort takes 6 comparisons

We could also instrument our comparison with a counter. Let's sort the student-list by candidate number:

```
(setq counter 0)

(defun icomp (a b)
  (progn (setq counter (+ counter 1)) (< (car a) (car b)))
)
```

## Trying it out

```
user> (mergesort studentlist icomp)

... big sorted list is printed ...

user> counter
341
user> (setq counter 0)
0
```

We can then check what happens with insertion sort and quicksort in the same way.

- ▶ Quicksort takes 884 comparisons.
- ▶ Insertion sort takes 1259 comparisons.

## More performance results

On a randomly generated list of 1000 integers:

- ▶ Quicksort took 24048 comparisons
- ▶ Mergesort took 8725 comparisons
- ▶ Insertion sort took 252248 comparisons.

## Some conclusions

Further experimentation with longer lists seems to confirm that

- ▶ mergesort uses the fewest comparisons of any of these algorithms
- ▶ insertion sort is disgustingly awful and should never be used
- ▶ our implementation of quicksort is a little heavy on its use of comparisons: note how the filters scan the entire list twice every time. A slightly improved version managed to sort the list of 1000 integers in 11038 comparisons rather than 24048.

In fact, mergesort is theoretically optimal, so it's no surprise that it comes out best in these tests.

## Why is it said that quicksort is better than mergesort?

Quicksort really begins to win over mergesort when it's implemented using *in-place array update* rather than our recursive, list-based version.

Comparisons are not the only key operation in sorting: creating the new list or array is costly too. Quicksort beats mergesort hands down on this front.

## Conclusions

- ▶ Algorithms are easy to code in functional style.
- ▶ Abstracting out details of algorithms makes code more generally applicable.
- ▶ Side-effects are a handy tool for some purposes, but are not in the spirit of functional programming.
- ▶ Lists are sometimes inefficient ways of representing data.