A Bubble Sort

Perhaps the best known sorting technique is the interchange or "bubble" sort. The idea is simple. We examine a list of values, looking for a pair of adjacent values that are "out of order." If we find such a pair, we swap the two values (placing them in correct order). Otherwise, the whole list must be in sorted order and we are done.

In conventional languages we need a lot of code to search for out-of-order pairs, and to systematically reorder them. In Prolog, the whole sort may be defined in a few lines:

```
bubbleSort(L,L) :- inOrder(L).
bubbleSort(L1,L2) :-
 append(X,[A,B|Y],L1), A > B,
 append(X,[B,A|Y],T),
 bubbleSort(T,L2).
```

The first line says that if **L** is already in sorted order, we are done.

The second line is a bit more complex. It defines what it means for a list **L2** to be a sorting for list **L1**, using our insight that we should swap out-of-order neighbors. We first partition list **L1** into two lists, **X** and **[A,B|Y]**. This “exposes” two adjacent values in **L**, **A** and **B**. Next we verify that **A** and **B** are out-of-order **(A>B)**. Next, in **append(X,[B,A|Y],T)**, we determine that list **T** is just our input **`L`**, with **`A`** and **`B`** swapped into **`B`** followed by **`A`**.

Finally, we verify that **`bubbleSort(T,L2)`** holds. That is, **`T`** may be bubble-sorted into **`L2`**.

This approach is rather more directed than our permutation sort—we look for an out-of-order pair of values, swap them, and then sort the “improved” list. Eventually there will be no more out-of-order pairs, the list will be in sorted order, and we will be done.

Merge Sort

Another popular sort in the "merge sort" that we have already seen in Scheme and ML. The idea here is to first split a list of length **L** into two sublists of length **L/2**. Each of these two lists is recursively sorted. Finally, the two sorted sublists are merged together to form a complete sorted list.

The bubble sort can take time proportional to n^2 to sort n elements (as many as $n^2/2$ swaps may be needed). The merge sort does better—it takes time proportional to $n \log_2 n$ to sort n elements (a list of size n can only be split in half $\log_2 n$ times).

We first need Prolog rules on how to split a list into two equal halves:

```
split([],[],[]).
split([A],[A],[]).
split([A,B|T],[A|P1],[B|P2]) :-
  split(T,P1,P2).
```

The first two lines characterize trivial splits. The third rule distributes one of the first two elements to each of the two sublists, and then recursively splits the rest of the list.

We also need rules that characterize how to merge two sorted sublists into a complete sorted list:

```
merge([],L,L).
merge(L,[],L).
merge([A|T1],[B|T2],[A|L2]) :-
   A =< B,  merge(T1,[B|T2],L2).
merge([A|T1],[B|T2],[B|L2]) :-
   A > B, merge([A|T1],T2,L2).
```

The first 2 lines handle merging null lists. The third line handles the case where the head of the first sublist is ≤ the head of the second sublist; the final rule handles the case where the head of the second sublist is smaller.

With the above definitions, a merge sort requires only three lines:

```
mergeSort([],[]).

mergeSort([A],[A]).

mergeSort(L1,L2) :-
   split(L1,P1,P2),
   mergeSort(P1,S1),
   mergeSort(P2,S2),
   merge(S1,S2,L2).
```

The first two lines handle the trivial cases of lists of length 0 or 1. The last line contains the full "logic" of a merge sort: split the input list, **`L`** into two half-sized lists **`P1`** and **`P2`**. Then merge sort **`P1`** into **`S1`** and **`P2`** into **`S2`**. Finally, merge **`S1`** and **`S2`** into a sorted list **`L2`**. That's it!

Quick Sort

The merge sort partitions its input list rather blindly, alternating values between the two lists. What if we partitioned the input list based on *values* rather than positions?

The *quick sort* does this. It selects a “pivot” value (the head of the input list) and divides the input into two sublists based on whether the values in the list are less than the pivot or greater than or equal to the pivot. Next the two sublists are recursively sorted. But now, after sorting, *no merge* phase is needed. Rather, the two sorted sublists can simply be appended, since we know all values in the first list are less than all values in the second list.

We need a Prolog relation that characterizes how we will do our partitioning. We we define **`partition(E,L1,L2,L3)`** to be true if **`L1`** can be partitioned into **`L2`** and **`L3`** using **`E`** as the pivot element. The necessary rules are:

```
partition(E,[],[],[]).
partition(E,[A|T1],[A|T2],L3) :-
  A<E, partition(E,T1,T2,L3).
partition(E,[A|T1],L2,[A|T3]) :-
  A>=E, partition(E,T1,L2,T3)
```

The first line defines a trivial partition of a null list. The second line handles the case in which the first element of the list to be partitioned is less than the pivot, while the final line handles the case in which the list head is greater than or equal to the pivot.

With our notion of partitioning defined, the quicksort itself requires only 2 lines:

```
qsort([],[]).
qsort([A|T],L) :-
  partition(A,T,L1,L2),
  qsort(L1,S1),qsort(L2,S2),
  append(S1,[A|S2],L).
```

The first line defines a trivial sort of an empty list.

The second line says to sort a list that begins with **`A`** and ends with list **`T`**, we partition **`T`** into sublists **`L1`** and **`L2`**, based on **`A`**. Then we recursively quick sort **`L1`** into **`S1`** and **`L2`** into **`S2`**. Finally we append **`S1`** to **`[A|S2]`**
(**`A`** must be > all values in **`S1`** and **`A`** must be $\leq$ all values in **`S2`**). The result is **`L`**, a sorting of **`[A|T]`**.

Arithmetic in Prolog

The = predicate can be used to test bound variables for equality (actually, identity).

If one or both of ='s arguments are free variables, = forces a binding or an equality constraint.

Thus

```
| ?- 1=2.
no
| ?- X=2.
X = 2
| ?- Y=X.
Y = X = _10751
| ?- X=Y, X=joe.
X = Y = joe
```

Arithmetic Terms are Symbolic

Evaluation of an arithmetic term into a numeric value must be *forced*.

That is, **`1+2`** is an infix representation of the relation **`+(1,2)`**. This term is *not* an integer!

Therefore

```
| ?- 1+2=3.
no
```

To force arithmetic evaluation, we use the infix predicate **`is`**.

The right-hand side of **`is`** must be all *ground terms* (literals or variables that are already bound). No *free* (unbound) variables are allowed.

Hence

```
|?- 2 is 1+1.
yes
| ?- X is 3*4.
X = 12
| ?- Y is Z+1.
! Instantiation error in argument
2 of is/2
! goal:  _10712 is _10715+1
```

The requirement that the right-hand side of an `is` relation be ground is essentially procedural. It exists to avoid having to invert complex equations. Consider,

```
(0 is (I**N)+(J**N)-K**N)), N>2.
```

Counting in Prolog

Rules that involve counting often use the **`is`** predicate to evaluate a numeric value.

Consider the relation **`len(L,N)`** that is true if the length of list **`L`** is **`N`**.

```
len([],0).
len([_|T],N) :-
   len(T,M), N is M+1.
| ?- len([1,2,3],X).
X = 3
| ?- len(Y,2).
Y = [_10903,_10905]
```

The symbols **`_10903`** and **`_10905`** are "internal variables" created as needed when a particular value is not forced in a solution.

Debugging Prolog

Care is required in developing and testing Prolog programs because the language is untyped; undeclared predicates or relations are simply treated as false.

Thus in a definition like

```
 adj([A,B|_]) :- A=B.
 adj([_,B|T]) :- adk([B|T]).
| ?- adj([1,2,2]).
no
```

(Some Prolog systems warn when an undefined relation is referenced, but many others don't).

Similarly, given

```
 member(A,[A|_]).
member(A,[_|T]) :-
  member(A,[T]).
| ?- member(2,[1,2]).
Infinite recursion!
```

(Why?)

If you're not sure what is going on, Prolog's trace feature is very handy.

The command

```
trace.
```

turns on tracing. (**notrace** turns tracing off).

Hence

```
| ?- trace.
yes
[trace]
| ?-  member(2,[1,2]).
```

```
    (1) 0 Call: member(2,[1,2]) ?
    (1) 1 Head [1->2]:
member(2,[1,2]) ?
    (1) 1 Head [2]:
member(2,[1,2]) ?
    (2) 1 Call: member(2,[[2]]) ?
    (2) 2 Head [1->2]:
member(2,[[2]]) ?
    (2) 2 Head [2]:
member(2,[[2]]) ?
    (3) 2 Call: member(2,[[]]) ?
    (3) 3 Head [1->2]:
member(2,[[]]) ?
    (3) 3 Head [2]: member(2,[[]])
?
    (4) 3 Call: member(2,[[]]) ?
    (4) 4 Head [1->2]:
member(2,[[]]) ?
    (4) 4 Head [2]: member(2,[[]])
?
    (5) 4 Call: member(2,[[]]) ?
```

Termination Issues in Prolog

Searching infinite domains (like integers) can lead to non-termination, with Prolog trying *every* value.

Consider

```
odd(1).
odd(N) :- odd(M), N is M+2.
| ?- odd(X).
X = 1 ;
X = 3 ;
X = 5 ;
X = 7
```

A query

```
?- odd(X), X=2.
```

going into an *infinite* search, generating each and every odd integer and finding none is equal to 2!

The obvious alternative,

`odd(2)` (which is equivalent to

`X=2, odd(X)`) also does an infinite, but fruitless search.

We’ll soon learn that Prolog does have a mechanism to “cut off” fruitless searches.

Definition Order can Matter

Ideally, the order of definition of facts and rules should not matter.

But,

in practice definition order can matter. A good general guideline is to define facts before rules. To see why, consider a very complete database of **`motherOf`** relations that goes back as far as

```
motherOf(cain,eve).
```

Now we define

```
isMortal(X) :-
  isMortal(Y), motherOf(X,Y).
isMortal(eve).
```

These definitions state that the first woman was mortal, and all individuals descended from her are also mortal.

But when we try as trivial a query as

```
| ?- isMortal(eve).
```

we go into an infinite search!

Why?

Let's trace what Prolog does when it sees

```
| ?- isMortal(eve).
```

It matches with the first definition involving **`isMortal`**, which is

```
isMortal(X) :-
  isMortal(Y), motherOf(X,Y).
```

It sets **`X=eve`** and tries to solve

```
isMortal(Y), motherOf(eve,Y).
```

It will then expand **`isMortal(Y)`** into

```
isMortal(Z), motherOf(Y,Z).
```

An infinite expansion ensues.

The solution is simple—place the "base case" fact that terminates recursion *first*.

If we use

```
isMortal(eve).
isMortal(X) :-
  isMortal(Y), motherOf(X,Y).
yes
| ?-  isMortal(eve).
yes
```

But now another problem appears!

If we ask

```
| ?-  isMortal(clarkKent).
```

we go into another infinite search!

Why?

The problem is that Clark Kent is from the planet Krypton, and hence won’t appear in our **motherOf** database.

Let’s trace the query.

It doesn’t match

```
isMortal(eve).
```

We next try

```
isMortal(clarkKent) :-
  isMortal(Y),
  motherOf(clarkKent,Y).
```

We try **Y=eve**, but **eve** isn’t Clark’s mother. So we recurse, getting:

```
isMortal(Z), motherOf(Y,Z),
motherOf(clarkKent,Y).
```

But **eve** isn’t Clark’s grandmother either! So we keep going further back, trying to find a chain of descendents that leads from **eve** to **clarkKent**. No such chain exists, and there is no limit to how long a chain Prolog will try.

There is a solution though!

We simply rewrite our recursive definition to be

```
 isMortal(X) :-
    motherOf(X,Y),isMortal(Y).
```

This is logically the same, but now we work from the individual **`X`** back toward **`eve`**, rather than from **`eve`** toward **`X`**. Since we have no **`motherOf`** relation involving **`clarkKent`**, we immediately stop our search and answer **`no`**!