Sorting Algorithms

Sorting algorithms are good examples of Prolog's definitional capabilities. In a Prolog definition the "logic" of a sorting algorithm is apparent, stripped of the cumbersome details of data structures and control structures that dominate algorithms in other programming languages.

Consider the simplest possible sort imaginable, which we'll call the "naive sort."

At the simplest level a sorting of a list `L` requires just two things:

- The sorting is a permutation (a reordering) of the values in `L`.
- The values are "in order" (ascending or descending).

We can implement this concept of a sort directly in Prolog. We
(a) permute an input list
(b) check if it is in sorted order
(c) repeat (a) & (b) until a sorting is found.

Permutations

Let's first look at how permutations are defined in Prolog. In most languages generating permutations is non-trivial—you need data structures to store the permutations you are generating and control structures to visit all permutations in some order.

In Prolog, permutations are defined quite concisely, though with a bit of subtlety:

`perm(X,Y)` will be true if list `Y` is a permutation of list `X`.

Only two definitions are needed:

```
perm([],[]).
perm(L,[H|T]) :-
  append(V,[H|U],L),
  append(V,U,W), perm(W,T).
```


The first definition,

```
perm([],[]).
```

is trivial. An empty list may only be permuted into another empty list.

The second definition is rather more complex:

```
perm(L,[H|T]) :-
append(V,[H|U],L),
 append(V,U,W), perm(W,T).
```

This rule says a list `L` may be permuted in to a list that begins with `H` and ends with list `T` if:

(1) `L` may be partitioned into two lists, `V` and `[H|U]`. (That is, `H` is somewhere in the "middle" of `L`).
(2) Lists `V` and `U` (all of `L` except `H`) may be appended into list `W`.
(3) List `W` may be permuted into `T`.

Let's see **perm** in action:

```
| ?- perm([1,2,3],X).
X = [1,2,3] ;
X = [1,3,2] ;
X = [2,1,3] ;
X = [2,3,1] ;
X = [3,1,2] ;
X = [3,2,1] ;
no
```

We'll trace how the first few answers are computed. Note though that *all* permutations are generated, and with no apparent data structures or control structures.

We start with **L=[1,2,3]** and **X=[H|T]**.

We first solve **append(V,[H|U],L)**, which

simplifies to **append(V,[H|U],[1,2,3])**.

One solution to this goal is

V = [], H = 1, U = [2,3]

We next solve **append(V,U,W)** which simplifies to **append([],[2,3],W)**.

The only solution for this is **W=[2,3]**.

Finally, we solve **perm(W,T)**, which simplifies to **perm([2,3],T)**.

One solution to this is **T=[2,3]**.

This gives us our first solution: **[H|T]=[1,2,3]**.

To get our next solution we *backtrack*. Where is the most recent place we made a choice of how to solve a goal?

It was at **perm([2,3],T)**. We chose **T=[2,3]**, but **T=[3,2]** is another solution. Using this solution, we get out next answer **[H|T]=[1,3,2]**.

Let's try one more. We backtrack again. No more solutions are possible for **perm([2,3],T)**, so we backtrack to an earlier choice point.

At **append(V,[H|U],[1,2,3])** another solution is

V=[1], H = 2, U = [3]

Using this binding, we solve **append(V,U,W)** which simplifies to **append([1],[3],W)**. The solution to this must be **W=[1,3]**.

We then solve **perm(W,T)** which simplifies to **perm([1,3],T)**. One solution to this is **T=[1,3]**. This

makes our third solution for **[H|T] = [2,1,3]**.

You can check out the other bindings that lead to the last three solutions.

A Permutation Sort

Now that we know how to generate permutations, the definition of a permutation sort is almost trivial.

We define an **inOrder** relation that characterizes our notion of when a list is properly sorted:

```
inOrder([]).
inOrder([_]).
inOrder([A,B|T]) :-
   A =< B, inOrder([B|T]).
```

These definitions state that a null list, and a list with only one element are always in sorted order. Longer lists are in order if the first two elements are in proper order. **(A=<B)** checks this and then the rest of the list,

excluding the first element, is checked.

Now our naive permutation sort is only one line long:

```
naiveSort(L1,L2) :-
  perm(L1,L2), inOrder(L2).
```

And the definition works too!

```
| ?-
naiveSort([1,2,3],[3,2,1]).
no
?- naiveSort([3,2,1],L).
L = [1,2,3] ;
no
| ?-
naiveSort([7,3,88,2,1,6,77,
 -23,5],L).
L = [-23,1,2,3,5,6,7,77,88]
```


Though this sort works, it is hopelessly inefficient—it repeatedly "shuffles" the input until it happens to find an ordering that is sorted. The process is largely undirected. We don't "aim" toward a correct ordering, but just search until we get lucky.

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 ≤ 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.