Notes for the last lecture of CSCI/ARTI 4540/6540, Fall 2012

Michael A. Covington

Finding the largest number in a list, and related problems

With pure logic:

```
biggest(List,Number) :-
        member(Number,List),
        \+ (member(NN,List), NN > Number).

?- biggest([5,4,9,4,6,3,3],What).
What = 9
```

This works, but it takes N^2 time.

Note that this works with backtracking alternatives – it does not have to see all the numbers at the same time.

```
age(sharon,4).
age(cathy,7).
age(danielle,5).

oldest(Name) :-
        age(Name,Age),
        \+ (age(_,AA), AA > Age).

?- oldest(Who).
Who = cathy
```

This still takes N^2 time.

By going through the list once and keeping track of the biggest so far:

```
biggest([A|Numbers], Result) :-
        biggest_aux(Numbers, A, Result).

biggest_aux([B|Numbers], BiggestSoFar, Result) :-
        B > BiggestSoFar,
        !,
        biggest_aux(Numbers, B, Result).

biggest_aux([_|Numbers], BiggestSoFar, Result) :-
        % we know that the first element is not > BiggestSoFar
        biggest_aux(Numbers, BiggestSoFar, Result).

biggest_aux([], Result, Result).

?- biggest([5,4,9,4,6,3,3], What).
What = 9
```

This takes N time; it only looks at each element once.

You must have all the alternative solutions in a list. If you don't, you can gather them using bagof.

Notice that this only gives us the biggest *number*. What if you want the *oldest child*? Then maybe define **older(Name1,Name2)**…

Getting a sorted list and looking for the first or last element:

This sometimes looks quick and simple, but constructing a sorted list takes $N \log N$ time.

How to pass your own comparison algorithm into one of these algorithms

The key idea is to use a 3-element list or 3-argument structure to give *two items to be compared and a term to be executed to compare them.*

[X, Y, X > Y] is a simple example.

Excursus: **\+ \+ Goal** will succeed iff **Goal** succeeds, but will not leave any instantiations.

```
best([A|Numbers], [X, Y, Comp], Result) :-
        best_aux(Numbers, [X, Y, Comp], A, Result).

best_aux([B|Numbers], [X, Y, Comp], BiggestSoFar, Result) :-
        \+ \+ (X = B, Y = BiggestSoFar, call(Comp)),
        !,
        best_aux(Numbers, [X, Y, Comp], B, Result).

best_aux([_|Numbers], [X, Y, Comp], BiggestSoFar, Result) :-
        % we know that the first element is not better than > BiggestSoFar
        best_aux(Numbers, [X, Y, Comp], BiggestSoFar, Result).

best_aux([], _, Result, Result).

6 ?- best([5,4,9,4,6,3,3],[X,Y,X>Y],What).
What = 9.

7 ?- best([5,4,9,4,6,3,3],[X,Y,X<Y],What).
What = 3.

8 ?- best([5,4,9,4,6,3,3],[X,Y,abs(5-X)<abs(5-Y)],What).
What = 5.
```

The things being compared need not be numbers. You could define any predicate that looks into them any way you need to.

Also, you could use this same technique to pass your own algorithm into another algorithm – such as a comparer into a sorting algorithm, or something entirely different.