Improving Efficiency by Helping

- Compute Fibonacci numbers
  F̋ib(1) = 1
  F̋ib(2) = 1
  F̋ib(N) = Fib(N-1) + Fib(N-2)

- Naive Algorithm using Definition

```
fib(1,1).
fib(2,1).
fib(N,F) :- N > 2,
    N1 is N - 1, fib( N1, F1 ),
    N2 is N - 2, fib( N2, F2 ),
    F is F1 + F2 .
```


Naive Fibonnaci Analysis

- Naive solution wastes effort by repeating parts of the computation

Use `assert` to Save Effort

- When each Fibonnaci Number is computed add it as a fact to the knowledge base.

```
fib( 1, 1 ).
fib( 2, 1 ).

fib( N, F ) :-
    N > 2 ,
    N1 is N - 1 , fib2( N1, F1 ),
    N2 is N - 2 , fib2( N2, F2 ),
    F is F1 + F2,
    asserta( fib( N, F ) ).
```


Using `assert` to Save Effort

- For a call of fib(N,F) all Fibonnaci numbers upto N will become facts in the knowledge base.

- The second recursive call in the definition will always succeed imediately because its results has been asserted in the knowledge base by the first recursive call.

Basic Bubble Sort

Algorithm: swap adjacent elements in the list until all items are in order

```
bubblesort( List, Sorted ) :-
    swap( List, List1 ),
    bubblesort( List1, Sorted ).

bubblesort( Sorted, Sorted ).

swap( [] , [] ).
swap( [ X, Y | Rest ], [ Y, X | Rest ]) :-
    gt( X, Y ).
swap( [ Z | Rest ] , [ Z | Rest1 ] ) :-
    swap( Rest, Rest1 ).
```


Use Helping Function for Efficiency

- Avoid recursive computation by computing Fibonnaci numbers in the forward direction.

```
fib( N, F ) :-
   forwardfib( 2, N, 1, 1, F ).
forwardfib( M, N, F1, F2, F2 ) :-
   M >= N .
forwardfib( M, N, F1, F2, F ) :-
   M < N ,
   NextM is M + 1 ,
   NextF2 is F1 + F2 ,
   forwardfib( NextM, N, F2, NextF2, F ).
```


Insertion Sort

- To sort a non-empty list `L = [X|T]`
  - Sort the tail T of L
  - Insert the head X into the sorted tail such that the resulting list is sorted

```
insertsort( [], [] ).
insertsort([ X | Tail ], Sorted ) :-
    insertsort( Tail, SortedTail ),
    insert( X, SortedTail, Sorted ).
insert( X, [ Y, Sorted ], [ Y | Sorted1 ] ) :-
    gt( X, Y ) , ! ,
    insert( X, Sorted, Sorted1 ).
insert( X, Sorted, [ X | Sorted ] ).
```


Sorting Lists

- Common Operation for Organizing Data
- Could use builtin list sorting primitives in Prolog. We'll look at how to do it directly.
- Assume that there is an ordering relation:
  `gt(X,Y)`
  for whatever data type we want to sort.
  Also assume non-empty lists.
- Look at several alternatives:
  bubble sort
  insertion sort
  quick sort

More Efficient List Concatenation

- Naive list concatenation:

```
conc( [], L, L ).
conc( [ X | L1 ], L2, [ X | L3 ] ):-
    conc( L1, L2, L3 ).
```

- This is very inefficient when first list is long.
  Recurs down first list until it is empty.
  Could do better if we could efficiently find end of first list.

- Change representation of lists to represent a list as a pair of lists.
  A *Difference List* represents a list L and a difference of two lists L1 and L2, written as L1-L2

Quicksort

To sort a non-empty list L:

- Select some element X from L as a *pivot*

- Split the rest of L into two lists (Small and Big) such that all elements of L greater than X are in Big and all the rest are in Small

- Sort Small to obtain SortedSmall
  Sort Big to obtain SortedBig

- The whole sorted list is the concatenation of
  SortedSmall and `[ X | SortedBig ]`

Difference List Example

```
Let   L = [ a, b, c ]

Then L can be represented as L1-L2 using any of

      L1 = [ a, b, c, d, e ]
      L2 = [ d, e ]

      L1 = [ a, b, c ]
      L2 = [ ]

      L1 = [ a, b, c, d, e | T ]
      L2 = [ d, e | T ]

Where T is any list.
Note that L2 is the end of the list L.
```


Quicksort Algorithm

```
quicksort([] , [] ).
quicksort([X | Tail ], Sorted) :-
    split(X, Tail, Small, Big),
    quicksort(Small, SortedSmall),
    quicksort(Big, SortedBig),
    conc(SortedSmall,[ X | SortedBig],Sorted).

split(X, [], [], []).
split(X, [ Y | Tail ], [ Y | Small ], Big):-
    gt( X, Y ), ! ,
    split( X, Tail, Small, Big ).
split(X, [ Y | Tail ], Small, [ Y | Big ]) :-
    split( X, Tail, Small, Big ).

conc( [], L, L ).
conc( [ X | L1 ], L2, [ X | L3 ] ):-
    conc( L1, L2, L3 ).
```

More Efficient Quicksort Using Difference Lists

```
quicksort( List, Sorted ) :-
   quicksort2(List,Sorted-[]).

quicksort2([],Z-Z).

quicksort2([ X | Tail], A1-Z2) :-
    split(X, Tail, Small, Big ),
    quicksort2(Small, A1-[X | A2]),
    quicksort2(Big, A2-Z2 ).
```