Lecture 2: Sorting Algorithms

Throughout this lecture, we'll assume a totally ordered type K of keys.

Sorting takes lists to lists.
We represent the input as data, but the output as codata:

sort :: List K → CoList K

One obvious structure is a fold:

isort = fold insert

insert :: ListF K (CoList K) → CoList K

The obvious next step is an unfold:

insert = unfold ins

ins :: ListF K (CoList K) → ListF K (ListF K (CoList K

It's not too difficult to complete the defn.

ins NilF = NilF

ins (ConsF a (Out° NilF)) = ConsF a NilF

ins ((ConsF a Out° (ConsF b x)))

| a≤b = ConsF a (ConsF b x)

| a>b = ConsF b (ConsF a x)

This definition is a bit clumsy, because it keeps recursing even after finding the right place to insert — we'll come back to that.

Another obvious place to start is on the structure of the output:

```
bsort = unfold bubble
bubble :: List K -> ListF K (List K)
```

Again, the next step is obvious:

```
bubble = fold bub
bub :: ListF K (ListF K (List K)) -> ListF K (List K)
```

And again, it's not too difficult to complete the definition:

```
bub NilF = NilF
bub (ConsF a NilF) = ConsF a (In NilF)
bub (ConsF a (ConsF b x))
    | a≤b = ConsF a (In (ConsF b x))
    | a>b = ConsF b (In (ConsF a x))
```

Note that this is bubblesort, not the selection sort you might expect: at each step, the least element is extracted as the next element of the output, but the remaining elements get rearranged - we'll come back to this too.

In fact, these two sorting algorithms are very closely related: one is a fold whose body is an unfold, the other is an unfold with body a fold and the inner bodies are identical apart from isomorphisms Out° ad In. If we define

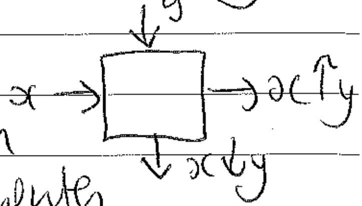
```
swap :: ListF K (ListF K α) → ListF K (ListF K α)
swap NilF = NilF
swap (ConsF a NilF) = ConsF a NilF
swap (ConsF a (ConsF b x))
    | a ≤ b = ConsF a (ConsF b x)
    | a > b = ConsF b (ConsF a x)
```

then

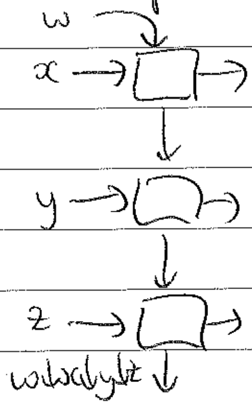
```
ins = swap · bimap id out
bub = bimap id In · swap
```

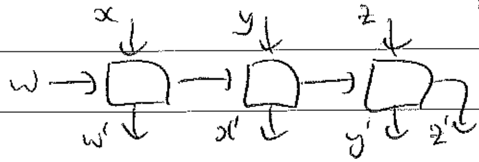
(Note that their earlier types are over-specific: they are both parametric in argument x.) Hinze et al (WGP 2012) explain that (CoList K, insert, out) and (List K, In, bubble) are both (ListF K)-bialgebras for distributive law swap.

The close correspondence between (naive) insertion sort and bubble sort can also be seen pictorially. Sorting networks can be built from 2×2 comparators.



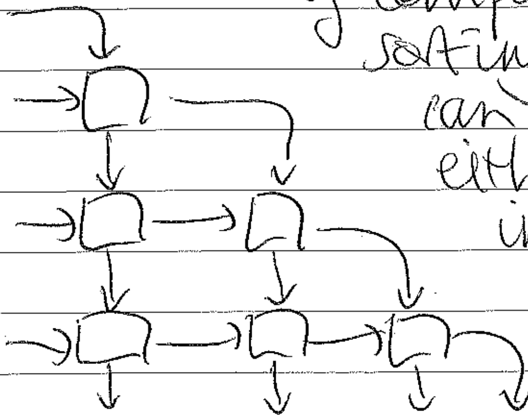
A column of these computes the minimum of a collection; a row inserts into a sorted sequence.





$x \le y \le z \Rightarrow w' \le x' \le y' \le z'$

Then a triangular array of comparators forms a sorting network, and can be viewed as either a stack of insertion or a pipeline of selections. (Thanks to MFP, Ruby, Lava.)



Recall bubblesort. To get selection sort instead, we need to select the minimum at each stage, but leave the remainder in same order. So when head is minimum, we should return tail unchanged. We have to preserve this tail to do so.

$fold :: Bifunctor\ f \Rightarrow (f\ \alpha\ \beta \to \beta) \to Mu\ f\ \alpha \to \beta$

$fold\ \varphi = \varphi \cdot bimap\ id\ (fold\ \varphi) \circ in^{\circ}$

Generalize to paramorphism:

$para :: Bifunctor\ f \Rightarrow (f\ \alpha\ (\beta \times Mu\ f\ \alpha) \to \beta) \to Mu\ f\ \alpha \to \beta$

$para\ \varphi = \varphi \cdot bimap\ id\ (\cancel{fold}\ para\ \varphi \vartriangle id) \circ in^{\circ}$

At each substructure, we make available both the original structure and its image under the computation. Then

$ssort = fold\ select$

$select = para\ sel$

$sel :: ListF\ K\ (ListF\ K\ (List\ K) \times List\ K) \to ListF\ K\ (List\ K)$

$sel\ (ConsF\ a\ (ConsF\ b\ x, y))$

$\quad |\ a \leq b = ConsF\ a\ y$

$\quad |\ a > b = ConsF\ b\ (In\ (ConsF\ a\ x))$

(other cases as for bub).

Dually, to get a smarter insertion sort, we should stop inserting when the right position is found - we need a way to leap straight to result.

unfold :: Bifunctor f ⇒ (β → f α β) → β → Nu f α

unfold φ = Out° ∘ bimap id (unfold φ) ∘ φ

Generalize to apomorphism:

apo :: Bifunctor f ⇒ (β → f α (β + Nu f α)) → β → Nu f α

apo φ = Out° ∘ bimap id (apo φ ▽ id) ∘ φ

For each substructure, we either get a seed for a corecursive call, or the result.

```
insert = apo smartins
smartins :: Listf K (Colist K) →
                  Listf K (Listf K (Colist K) + Colist K)
smartins (Consf a (Out° (Consf b x)))
    | a≤b = Consf a (Right (Out° (Consf b x)))
    | a>b = Consf b (Left (Consf a x))
smartins (Consf a (Out° Nilf))
            = Consf a (Left Nilf)
     or     = Consf a (Right (Out° Nilf))
smartins Nilf = Nilf
```