# 6 Sorting

To sort a list, the elements of the list must be orderable, and sorting amounts to finding a permutation of the list that is sorted.

```
> sort :: Ord a => [a] -> [a]
> sort = head . filter sorted . permutations
```

A list is *sorted* if every adjacent pair of elements of the list is in the right order.

```
> sorted :: Ord a => [a] -> Bool
> sorted = all ordered . pairs where ordered = uncurry (<=)
```

Here $all\ p = and \cdot map\ p$ is predefined, as is *and* which is the fold of (&&), and *uncurry* turns a function of two arguments into one that takes a pair.

There is a handy idiom for the adjacent pairs of a list

```
> pairs :: [a] -> [(a, a)]
> pairs xs = xs `zip` tail xs
```

which takes two copies of the list, shifts one along by a place, and makes pairs of corresponding elements with the predefined function *zip* which might have been (but is not in fact) defined by

```
> zip :: [a] -> [b] -> [(a, b)]
> zip (x:xs) (y:ys) = (x,y) : zip xs ys
> zip    _      _   = []
```

Notice that it discards any of either list that is left over after reaching the end of the other.

What about the *permutations* of a list? One natural definition considers all the permutations that begin with each element of the list:

```
> permutations [] = [[]]
> permutations xs =
>         [ x:ys | x <- xs, ys <- permutations (exclude x xs) ]
```

Unfortunately excluding an element requires a test for equality

```
> exclude y xs =
>           takeWhile (/= y) xs ++ tail (dropWhile (/= y) xs)
```

so this definition gives a function whose type needs *Eq* on the type of elements. (The need for *Eq* can be avoided by generating $x$ and $exclude\ x\ xs$ at the same time, but doing that is a little involved and left as an exercise.)

An alternative definition considers all permutations of the tail of the list, and inserts the head into each possible position in the list.

```
> permutations' :: [a] -> [[a]]
> permutations' [] = [[]]
> permutations' (x:xs) =
>          [ zs | ys <- permutations' xs, zs <- include x ys ]

> include :: a -> [a] -> [[a]]
> include x    []  = [[x]]
> include x (y:ys) = (x:y:ys) : map (y:) (include x ys)
```

Of course the result is that the premutations are in a different order, but the order does not matter here.

The only problem with this definition of *sort* is that it takes infeasibly long for quite small inputs because of the huge number of permutations. There are $n!$ permuations of $n$ things, and in the worst case *sort* must check them all, so takes at least a time proportional to $n!$. Sorting a list of twenty things takes $20!/10! \approx 6 \times 10^{11}$ times as long as sorting ten things.

## 6.1 Insertion sort

One way to think of a better strategy is to imagine picking up and sorting a hand of cards. Many players will keep a partial hand in sorted order, inserting each new card into its proper place.

```
> isort :: Ord a => [a] -> [a]
> isort    []  = []
> isort (x:xs) = insert x (isort xs)
```

This is of course another instance of *fold*:

```
> isort' :: Ord a => [a] -> [a]
> isort' = fold insert []
```

Inserting an element into a sorted list is a matter of finding the right place:

```
> insert :: Ord a => a -> [a] -> [a]
> insert x xs = takeWhile (<x) xs ++ [x] ++ dropWhile (<x) xs
```

Notice that on every call of *insert* the *xs* will be sorted, so in consequence the result will also be sorted (and so the result of *isort* will be sorted).

Since *insert* is in the worst case linear in the length of the list into which it inserts, and is called a number of times linear in the lenth of the input, insertion sort takes an effort quadratic in the length of the input.

## 6.2 Selection sort

Selection sort works in the opposite way: the sorted hand is built up in order by extracting the smallest element (which is the head of the answer) and then proceeding to sort the remainder.

```
> ssort :: Ord a => [a] -> [a]
> ssort [] = []
> ssort xs = y : ssort ys
>            where y = minimum xs
>                  ys = exclude y xs
```

The predefined function *minimum* is a fold (on non-empty lists), and we met *exclude* earlier. However *ssort* is itself not obviously a fold; it is however an instance of an *unfold*.

The (non-standard) function *unfold* is in a sense dual to *fold*. Lists are constructed from the constructors (:) and [] and it is a property of folds that $fold\ (:)\ []\ =\ id$. Dually, lists are deconstructed by *head* and *tail* having first checked whether they are *null*. The dual of $fold\ (:)\ []\ =\ id$ is that $unfold\ null\ head\ tail = id$. Provided that

$$\begin{aligned} p\ n &= True \\ p\ (c\ x\ y) &= False \\ h\ (c\ x\ y) &= x \\ t\ (c\ x\ y) &= y \end{aligned}$$

it can be proved that $unfold\ p\ h\ t \cdot fold\ c\ n = id$.

One way of defining *unfold* would be

```
> unfold :: (a->Bool) -> (a->b) -> (a->a) -> a -> [b]
> unfold null head tail =
>           map head . takeWhile (not.null) . iterate tail
```

where the (predefined) function

```
iterate :: (a->a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

produces the infinite list $[x, f\ x, f(f\ x), f(f(f\ x)), \ldots]$. So for example

```
*Main> unfold (==0) (`mod`10) (`div`10) 123456
[6,5,4,3,2,1]
```

Selection sort can be written as an unfold

```
> ssort' = unfold null minimum excludeMin
>          where excludeMin xs = exclude (minimum xs) xs
```

showing that it is a mechanism in this sense dual to insertion sort.

This definition is less efficient than the direct recursion, since it has to calculate the *minimum* twice for each tail of the input. (This is an artefact of our definition of *unfold* and can be eliminated.)

However we do it, selection sort is similarly quadratic in the length of the input. You might know, or might perhaps learn next term that a sorting function with the type of *sort* has in the worst case to take at least $O(n \log n)$ work on a list of length $n$.

## 6.3 Quicksort

*Quicksort* was invented in about 1960 by Tony Hoare. He had been a computer salesman, and perhaps the very best thing about Quicksort is the compelling (and not entirely inaccurate) name.

This is a particular case of the *divide and conquer strategy* for breaking down big problems into smaller ones, until there are only easy problems left. The idea is to pick a value from the list, and to divide the rest of the list into smaller and larger elements. These can then be sorted independently and the sorted answer is just the concatenation of the sorted sublists.

```
> qsort :: Ord a => [a] -> [a]
> qsort    []  = []
> qsort (x:xs) = qsort [ y | y <- xs, y < x  ] ++
>                  x : [ y | y <- xs, y == x ] ++
>                qsort [ y | y <- xs, y > x  ]
```

This recursion works because each of the calls on the right-hand side has an argument shorter than the one on the left-hand side.

The idea is that if you are lucky, the so called pivot $x$ will divide the list in half and the recursion never becomes more than $\log n$ deep, so the total amount of work never exceeds $O(n \log n)$. Unfortunately if you happen to have a pivot which is an extreme value (because the input was already sorted, or reverse-sorted) this algorithm is also quadratic.

You will find that presentations of Quicksort normally involve elements of arrays, and are full of array indexes. (In fairness, they are achieving more than our version because they sort the array *in place* and often access the array in fairly predictable ways, both of which matter if you have a huge collection to sort and a cached memory system.) One of the beauties of functional programming is the ability to present an algorithm like this in a more abstract way that clearly shows its structure.

## 6.4 Merge sort

In order to guarantee worst-case performance as good as $O(n \log n)$ a divide-and-conquer algorithm has to guarantee a fairly even split into subtasks. One such algorithm is *merge sort* probably invented about 1945 by John von Neumann. Provided there are at least two elements in the list, it can be divided into two smaller lists which can be sorted recursively, reducing the original problem into one of merging two sorted runs into one.

```
> msort :: Ord a => [a] -> [a]
> msort []  = []
> msort [x] = [x]
> msort xs  = merge (msort ls) (msort rs)
>             where (ls,rs) = halve xs
```

Merging two runs can be done by a recursion not entirely unlike that for *zip*

```
> merge :: Ord a => [a] -> [a] -> [a]
> merge (x:xs) (y:ys) | x <= y    = x : merge xs (y:ys)
>                     | otherwise = y : merge (x:xs) ys
> merge    []     ys  = ys
> merge    xs     []  = xs
```

This *merge* is linear in the size of the result, so if *halve* guarantees two half-sized sub-problems, *msort* is guaranteed asymptotically optimally even in the worst case.

As written, *merge* breaks ties between equal values by putting the value from the left run first, so if *halve* divides a list into an initial segment and a final segment the resulting *msort* will be *stable*: it does not permute those values which compare as equal to each other.

Finally

```
> halve xs = (take n xs, drop n xs) where n = length xs `div` 2
```

although this version unnecessarily makes three passes over much of the list.

## 6.5 Some small optimisations

As happened in this lecture, often when one wants *take n xs* one also needs *drop n xs* for the same *n* and *xs*. It seems wasteful to compute both, since the computation of one necessarily discovers the other. There is a predefined function

```
> splitAt :: Int -> [a] -> ([a], [a])
```

which satisfies

```
> splitAt n xs = (take n xs, drop n xs)
```

although that is of course not the defining equation. So

```
> halve xs = splitAt (length xs `div` 2) xs
```

makes only two passes through the list. I chose to use the separate functions for clarity.

Similarly there is a function

```
span :: (a -> Bool) -> [a] -> ([a], [a])
```

which satisfies

```
span p xs = (takeWhile p xs, dropWhile p xs)
```

You might expect that the point of these paired functions was to save time, and if applying $p$ is expensive that might be the case. If it is cheap however, the extra book-keeping means that not much time is saved.

The real point of *span* is that it saves space.

Separate computations of *takeWhile p xs* and *dropWhile p xs* have to happen in some order or another. While one is happening, the whole of $xs$ has to be kept in existence for the as yet unevaluated computation of the other. However a call of *span* can discard parts of $xs$ as they are used up in the *takeWhile* result, since it knows they are not needed in the *dropWhile*.