# Lecture 11: Map-Reduce, Lists and Sorting

# Higher Order Operators on lists

- recall sum/max for a list. Notice that both functions look almost identical. With the exception of the different types and different operation (+,max), both functions are equivalent.  In both cases, we walk down a list performing some operation with the data at each step.

```
fun max(a,b) = if a>b then a else b;
fun maxm(list)=
     let
          fun max_it (cur_max, []) = cur_max
          | max_it (cur_max, (x::xs)) = max_it(max(cur_max,x),xs)
     in
       max_it(hd(list),list)
end;
```

- in the iterative version we use an accumulator (eg. cur_max)

# Reduce

- can we generalise that
  - consider a binary operator ⊙ (eg max(a,b) or sum(a,b)=a+b)
  - let $L=[l_1, l_2, l_{3},......, l_n]$
    - The higher order function reduce is to compute $l_1 \odot l_2 \odot l_3 \odot ...... \odot l_n$
    - to make it iterative reduce needs to have 3 parameters —
    - *f, init_acc, list* (here f(a,b) computes a ⊙ b)

**The operator ⊙ need not be associative**
**Right or Left Associative?**

- so what should we compute?

$$((((((l_1 \odot l_2) \odot l_3) \odot \ldots\ldots \odot l_n) \quad \text{or} \quad (l_1 \odot (l_2 \odot (l_3 \odot \ldots\ldots \odot l_n))))))$$

- sml has two versions of reduce called foldl and foldr

`foldl f init [x1, x2, ..., xn]`
returns
`f(xn,...,f(x2, f(x1, init))...)`
or *init* if the list is empty.

`foldr f init [x1, x2, ..., xn]`
returns
`f(x1, f(x2, ..., f(xn, init)...))`
or *init* if the list is empty.

# foldl and foldr

```
val l=[1,2,7,2,18,12,~2];
foldr max (hd(l)) l;
fun sum(a,b)=a+b;
foldr sum 0 l;
```

```
> val l = [1, 2, 7, 2, 18, 12, ~2]: int list;
> val it = 18: int;
> val sum = fn: int * int → int;
> val it = 40: int;
```

```
foldl sum 0 l;
foldl max (hd(l)) l;
```

```
> val it = 40: int;
> val it = 18: int;
```

# curried map

- ML chooses the most general (least-restrictive) type possible for user-defined functions.
- The function definition fun f x y = expression; defines a function f (of x) that returns a function (of y). Reducing multiple argument functions to a sequence of one argument functions is called currying (after Haskell Curry, a mathematician who popularized the approach)
- In curried form map is defined as

```
fun map f [] = []
    | map f (x::y) = (f x) :: map f y;
val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

  - given f: α —> α
  - val foo=map(f) gives a curried function foo from α-list —> α-list
  - foo(L) will return list with f applied on each element of L i.e equivalent to original map(f,L)

# combining map & reduce (map-reduce)

```
val l=[1,2,7,2,18,12,~2];
foldr max (hd(l)) l;
fun sum(a,b)=a+b;
foldr sum 0 l;
```

```
fun square(x)= x*x;
val sq=map(square);
sq(l);
foldl sum 0 (sq(l));
foldl max 0 (sq(l));
```

```
> val square = fn: int → int;
> val sq = fn: int list → int list;
> val it = [1, 4, 49, 4, 324, 144, 4]: int list;
> val it = 530: int;
> val it = 324: int;
```

# Insertion into Sorted List

- Sorted List L=[1,2,3,7,8,11,12]
- Insert 9 into L —> we need to ensure new list is sorted

```
l=[1,2,7,12,18,122]
insert(9,l)
                9,  [1,2,7,12,18,122]
    1::         9,  [2,7,12,18,122]
    1::2::      9,  [7,12,18,122]
    1::2::7::   9,  [12,18,122]
                  | [9,12,18,122]
```

# More with Lists: Insertion into a sorted list

**Example 7.8** *Inserting an element into a sorted list.*
We will develop the function of the type $insert : \alpha \times \alpha\text{–}LIST^{sorted} \rightarrow \alpha\text{–}LIST^{sorted}$, where $\alpha\text{–}LIST^{sorted}$ is the data-type denoting all lists sorted in the ascending order. We can develop the function *insert* by inducting on the length of the list. The base case is clearly given as `insert(a,[]) = a::[]`. Given the inductive hypothesis that we can solve the problem `insert(a,ls)`, we can program the induction step as follows

$\langle insert \rangle \equiv$

```
fun insert(a,[]) = [a]
  | insert(a,x::ls) =
      if a < x then a::x::ls else x::insert(a,ls);
```

# Insertion into sorted list: Example

```
fun insert(a,[]) = [a]
  | insert(a,x::ls) =
     if a < x then a::x::ls else x::insert(a,ls);
```

```
l=[1,2,7,12,18,122]
insert(9,l)
               9, [1,2,7,12,18,122]
    1::        9, [2,7,12,18,122]
    1::2::     9, [7,12,18,122]
    1::2::7::  9, [12,18,122]
                 [9,12,18,122]
```

Result = [1,2,7,9,12,18,122]

Complexity: worst case recursion depth n hence O(n)

Exercise: Prove correctness of insert using Induction!

# Merge two sorted lists

**Example 7.9** *Merging two sorted lists.*
We can again develop an algorithm for the function $merge : \alpha\text{–}LIST^{sorted} \times \alpha\text{–}LIST^{sorted} \rightarrow \alpha\text{–}LIST^{sorted}$ inductively. Inducting on the length of the lists, we have the base cases `merge([],l2) = l2` and `merge(l1,[]) = l1`. Given that we can solve `merge(l1,y::l2)` when `l1` is of size $n \geq 0$ and `merge(x::l1,l2)` when `l2` is of size $m \geq 0$, we can write the program as

$\langle$*Code for merge*$\rangle \equiv$

```
fun merge([],l2) = l2
  | merge(l1,[]) = l1
  | merge(x::l1,y::l2) =
        if x <= y then x::merge(l1,y::l2) else y::merge(x::l1,l2);
```

# List Merge Example & Complexity

```
l=[1,2,7,12]  l2=[3,8,11]
merge(l,l2)          =[1,2,7,12]      [3,8,11]
                      1<3,  [2,7,12]      [3,8,11]
  1::                 2<3   [7,12]        [3,8,11]
  1::2::3::           3<7   [7,12]        [8,11]
  1::2::3::7::        7<8   [12]          [8,11]
  1::2::3::7::8        8<12   [12]        [11]
  1::2::3::7::8::11          [12]         []
                             return [12]
1::2::3::7::8::11:: [12]= [1,2,3,7,8,11,12]
```

- Complexity:
    - if lists are length m & n merge is O(m+n)

# Sort a list: Sort by repeated insertion of elements

## Insertion Sort

- sort([4,0,9,2]) => insert(4,insert(0,insert(9,insert(2,[]))))

**Example 7.10** *Insertion sort.*
We will define a function $insort : \alpha\text{–}LIST \rightarrow \alpha\text{–}LIST^{sorted}$, in terms of the function $insert$ defined in Example 7.8 as follows. Inducting on the length of the list, the base case is clearly `insort([]) = []`. Given that we can solve `insort(ls)`, the problem `insort(x::ls)` is merely the problem of inserting `x` in to the sorted list `insort(ls)`. Hence, we have

$\langle insort \rangle \equiv$

```
fun insort([]) = []
  | insort(x::ls) = insert(x,insort(ls));
```

# Time Complexity of Insertion Sort

- let T(n) be the maximum possible time required for insort() on a list L of n elements
- T(n) <= T(n-1)+n
    - recursive call for tl(L) requires at most T(n-1)
    - the insert of hd(L) can require at most n steps
- $T(n) <= n+n-1+T(n-2) <= n+n-1 \ldots +T(0) = n(n+1)/2 = O(n^2)$

# Faster Ways of Sorting

- Strategy: Divide & Conquer
    - Try and divide the list into two roughly equal parts
    - sort each part recursively
    - combine the result to produce final sorted list
- If divide & combine can be done in $O(n)$ can this improve complexity?
    - $T(n) = 2\ T(n/2) + cn = 4\ T(n/4) + cn + cn$ …..
    - $T(2^k) = 2^k T(1) + kcn <= O(n \log n)$ since $k=\log_2 n$

# Division Strategy

```
[— — — — — — — — — — — — — — -]
[— — — — — — —]   [— — — — — — —]
[— — —]   [— — —]   [— — —]   [— — —]
[— -]  [— -]  [— -]  [— -]  [— -]  [— -]  [— -]  [— -]
[-] [-] [-] [-] [-] [-] [-] [-] [-] [-] [-] [-] [-] [-] [-] [-]
```

- O(n) work at each level, log n recursive depth —> O(n log n)

# Division Strategy — Unequal division

[— — — — — — — — — — — — — -]

[— — — — — — — — —]   [— — — —]

[— — — —]   [— — — —]   [— —]   [— —]

[— —]  [— —]  [— —]  [— —]   [-]  [-]  [-]  [-]

[-]  [-]  [-]  [-]  [-]  [-]  [-]  [-]  [-]  [-]

- max cn work at each level * recursive depth (d) —> O(nd)
- if d is O(log n) this is still O(log n)
- eg if division is 1/4, 3/4 the longest list after k recursive steps is $n(3/4)^{k}=1$ when $k = \log_{1.33} n$
- still O(n log n) since $\log_{1.33} n = \log_2 n / \log_2 1.33$

# Merge Sort

- Divide the input list into two equal lists (split)
- recursively sort L1, L2 and then use merge to merge the sorted lists

```
fun msort([]) = []
  | msort(x::[]) = x::[]
  | msort(L) =
let <code for split, merge>
        val (L1,L2) = split(L)
    in
        merge( msort(L1), msort(L2) )
end;
```

# Complexity of Mergesort

- depth is log n
- split & merge each can take O(n)
- hence O(n log n)
- what about space complexity?

# QuickSort

- let p=hd(l)
- split L into to L1= elements in L <=p, L2= elements in L > p
- [10,2,7,12,28,22] —> p=10 [2,7] [12, 28, 22]
- sort the lists recursively — let them return S1 & S2
- return S1@ (p::S2)
    - in this ex R1=[2,7], R2= [12,22,28]
    - [2,7] @ [10,12,22,28] = [2,7,10,12,22,28]

# Quicksort

```
fun qsort([]) = []
   | qsort(x::xs) =
      let
          fun comp opr x y = opr(y,x):bool
      in
          qsort(x::xs) = qsort(List.filter (comp op<= x) xs) @ (x::qsort(List.filter (comp op> x) xs));
     end;
```

```
> val qsort = fn: int list → int list;
> val it = [1, 2, 7, 2, 18, 12, ~2]: int list;
> val it = [~2, 1, 2, 2, 7, 12, 18]: int list;
```

# Complexity of Quicksort

- split & append each can take O(n)
- what is the depth? if depth O(d) complexity is O(n d)
- depth depends on how many elements in L1 & L2? Do they divide evenly?
- give example of worst case?  [hint input is almost sorted]
- what about space complexity?