15-150
Spring 2018
Dilsun Kaynar
LECTURE 7
Sorting Integer Lists
**comparison**

```plaintext
compare : int * int -> order

datatype order = LESS | EQUAL | GREATER

fun compare(x:int, y:int):order =
    if x<y then LESS
    else
        if y<x then GREATER
        else EQUAL

compare(x,y) = LESS             if x<y
compare(x,y) = EQUAL         if x=y
compare(x,y) = GREATER     if x>y
```
**insertion**

\[ \text{ins} : \text{int} \times \text{int list} \rightarrow \text{int list} \]

(* REQUIRES: L is a sorted list *)

(* ENSURES: \( \text{ins}(x, L) \) is a sorted permutation of \( x::L \) *)

\[
\begin{align*}
\text{fun} & \quad \text{ins} \ (x, \ [ \ ]) = \ [x] \\
& \quad \text{ins} \ (x, \ y::L) = \ \text{case} \ \text{compare}(x, \ y) \ \text{of} \\
& \quad \quad \quad \text{GREATER} \Rightarrow y :: \ \text{ins}(x, \ L) \\
& \quad \quad \quad \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \Rightarrow x :: y :: L
\end{align*}
\]
isort

isort : int list -> int list

(* REQUIRES:  true *)

(* ENSURES:  isort(L) is a sorted perm of L *)

fun isort [ ] = [ ]
  |
  isort (x::L) = ins (x, isort L)
Let $W_{\text{ins}}(n)$ be the work for $\text{ins}(x, L)$ when $x, L$ are values and $L$ has length $n$.

$W_{\text{ins}}(n)$ is $O(n)$.

Let $W_{\text{isort}}(n)$ be the work for $\text{isort}(L)$ when $L$ is a list of length $n$.

$W_{\text{isort}}(0) = 1$
$W_{\text{isort}}(n) = 1 + W_{\text{ins}}(n-1) + W_{\text{isort}}(n-1)$ for $n > 0$

$W_{\text{isort}}(n)$ is $O(n^2)$.
Goals for today

• State how merge sort algorithm works
• Write code for merge sort
• Observe opportunities for parallelism
• Write recurrences for our code and find big-O bounds for work and span
mergesort

A recursive *divide-and-conquer* algorithm

- If list has length 0 or 1, do nothing.
- If list has length 2 or more,
  
  *split* the list into two shorter lists,
  *sort* these lists,
  *merge* the results

(not a good specification of input-output behavior, but does describe an *algorithm*)
(* REQUIRES: true *)

(* ENSURES: msort(L) is a sorted perm of L *)

fun msort ([ ] : int list) : int list = [ ]

| msort [x] = [x]

| msort L =

  let
  val (A, B) = split L
  in
  merge (msort A, msort B)
  end

msort [4,2,1,3] ==> [1,2,3,4]
**split**

split : int list -> int list * int list

(* REQUIRES: true
ENSURES:split(L) = a pair of lists (A, B) such that length(A) and length(B) differ by at most 1, and A@B is a permutation of L. *)

```ocaml
fun split ([ ] : int list) : int list * int list = ([ ], [ ])
  | split [x] = ([x], [ ])
  | split (x::y::L) =
    let val (A, B) = split L in (x::A, y::B) end
```
**merge**

*REQUIRES: A and B are sorted lists*

*ENSURES: merge(A, B) = a sorted perm of A@B*

```ml
fun merge (A : int list, [ ] : int list) : int list = A
| merge ([ ], B) = B
| merge (x::A, y::B) = case compare(x, y) of
  LESS  => x :: merge(A, y::B)
  | EQUAL  => x :: y :: merge(A, B)
  | GREATER => y :: merge(x::A, B)
```
msort

msort : int list -> int list

(* REQUIRES: true *)
(* ENSURES: msort(L) is a sorted perm of L *)

fun msort ([ ] : int list) : int list = [ ]
| msort [x] = [x]
| msort L =

  let
  val (A, B) = split L
  in
  merge (msort A, msort B)
  end

msort [4,2,1,3] ==> [1,2,3,4]
fun split ([ ]) : = ([ ], [ ])
|   split [x] = ([x],[ ])
|   split (x::y::L) =
|       let val (A, B) = split L
|       in (x::A, y::B) end

fun merge (A : int list, [ ] : int list) : int list = A
|   merge ([ ], B) = B
|   merge (x::A, y::B) = case compare(x, y) of
|       LESS => x :: merge(A, y::B)
|       EQUAL => x :: y :: merge(A, B)
|     GREATER  => y :: merge(x::A, B)

\[ W_{\text{split}}(n) : \text{work of split}(L) \text{ when length}(L)=n \]

\[ W_{\text{split}}(n) \text{ is } O(n) \]

\[ W_{\text{merge}}(a,b) : \text{work of merge}(A,B) \text{ when length}(A)=a \& \text{length}(B)=b \]

\[ W_{\text{merge}}(a,b) \text{ is } O(a+b) \]
\[ W_{\text{msort}}(n) \text{ work of } \text{msort}(L) \text{ when length}(L)=n \]

\[ W_{\text{msort}}(0) = c_0 \]

\[ W_{\text{msort}}(1) = c_1 \]

\[ W_{\text{msort}}(n) = c_2 + W_{\text{split}}(n) + 2W_{\text{msort}}(n \div 2) \text{ for } n > 1 \]

\[ + W_{\text{merge}}(n \div 2, n \div 2) \]

\[ = O(n) + 2W_{\text{msort}}(n \div 2) \]

\[ \leq c_3 \cdot n + 2W_{\text{msort}}(n \div 2) \]

\[ W_{\text{msort}}(n) \text{ is } O(n \log n) \]
• msort(L) does $O(n \log n)$ work, where $n$ is the length of $L$

• List operations are inherently *sequential*
  
  • $e_1 :: e_2$ evaluates $e_1$ first, then $e_2$
  
  • *split* and *merge* are not easily *parallelizable*

• We *could* use parallel evaluation in msort(L) for the recursive calls to msort A and msort B

*How would this affect runtime?*
\begin{align*}
\text{msort } L & = \textbf{let} \ \textbf{val} \ (A, \ B) = \text{split } L \ \textbf{in} \\
& \quad \text{merge (msort } A, \ \text{msort } B) \ \textbf{end} \\
\text{span} & \\
S_{\text{msort}}(n) & = c + S_{\text{split}}(n) + S_{\text{merge}}(n) + \\
& \quad \max(S_{\text{msort}}(n \ \text{div} \ 2), \ S_{\text{msort}}(n \ \text{div} \ 2)) \\
\leq k \cdot n + S_{\text{msort}}(n \ \text{div} \ 2) \\
\end{align*}

\text{S_{msort}(n) is O(n)}
Trees offer better support for parallel evaluation
trees

data type tree = Empty | Node of tree * int * tree

• A user-defined type named tree
• With constructors Empty and Node

  Empty : tree
  Node : tree * int * tree -> tree
Empty is a sorted tree

Node(l, x, r) is a sorted tree iff

- every integer in l is $\leq x$,
- every integer in r is $\geq x$,
- and l, r are sorted trees
Divide and conquer

• Split the tree into subtrees
• Sort the subtrees
• Merge the results
**Msort**

Msort : tree -> tree

**REQUIRES:** true

**ENSURES:** Msort(t) is a sorted tree consisting of the items of t

```ml
fun Msort Empty = Empty

| Msort (Node(l, x, r)) =
  | Ins (x, Merge(Msort l, Msort r))
```
fun Ins (x, Empty) = Node(Empty, x, Empty)
| Ins (x, Node(l, y, r)) =
    case compare(x, y) of
    | GREATER => ______________________
    | _        => ______________________
    | _        => _______________________
tree insertion

Ins : int * tree -> tree
REQUIRES: t is a sorted tree
ENSURES: Ins(x,t) is a sorted tree consisting of x and all of t

fun Ins (x, Empty) = Node(Empty, x, Empty)
| Ins (x, Node(l, y, r)) =
  case compare(x, y) of
    GREATER => Node(l, y, Ins(x, r))
  | _        => Node(Ins(x, l), y, r)
fun Ins (x, Empty) = Node(Empty, x, Empty)
  | Ins (x, Node(t1, y, t2)) =
    case compare(x, y) of
      GREATER => Node(t1, y, Ins(x, t2))
    | _        => Node(Ins(x, t1), y, t2)

Ins(4, 3)

=>

3

1

6

2 5

Ins(4, 6)

=>

3

1

Ins(4, 6)

2

5

Ins(4, 5)