today

Sorting an integer list

• specifications and proofs
• asymptotic analysis

SML features
• datatype definitions
• boolean connectives
• case expressions
• <> means ≠
comparison

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
properties of $\leq$

• $\leq$ is a *linear ordering*

For all values $a,b,c : \text{int}$

- If $a \leq b$ and $b \leq a$ then $a = b$ (antisymmetric)
- If $a \leq b$ and $b \leq c$ then $a \leq c$ (transitive)
- Either $a \leq b$ or $b \leq a$ (total, or linear)

• $<$ is defined by

  $a < b$ if and only if ($a \leq b$ and $a \neq b$)

and satisfies

  $a < b$ or $b < a$ or $a = b$ (trichotomy)
A list of integers is \(-\text{sorted}\) (or just \(\text{sorted}\))
if each item in the list is \(\leq\) all later items.

\[
\text{fun } \text{sorted} \; [ \; ] = \text{true} \\
| \text{sorted} \; [x] = \text{true} \\
| \text{sorted} \; (x::y::L) = \\
\quad (x \leq y) \; \text{andalso} \; \text{sorted}(y::L)
\]

For all \(L : \text{int list}\),
\[
\text{sorted}(L) = \text{true} \; \text{if } L \; \text{is } \text{sorted} \\
= \text{false} \; \text{otherwise}
\]
A list of integers is \(-\text{sorted}\) (or just \text{sorted}) if each item in the list is \(\leq\) all later items.

\[
\begin{align*}
\text{fun } \text{sorted} \ [ \ ] &= \text{true} \\
| \text{sorted} \ [x] &= \text{true} \\
| \text{sorted} \ (x::y::L) &= (\text{compare}(x,y) \nless \text{GREATER}) \text{ andalso } \text{sorted}(y::L)
\end{align*}
\]}

An equivalent definition, using \text{compare}

\text{order is an equality type}
specs and code

• We use `sorted` only in `specifications`.
• Our sorting functions won’t `use` it.
• But you `could` use it for testing...
insertion sort

- **Insertion sort** is a simple *sorting algorithm* that builds the sorted list recursively, one item at a time.
- If the list is empty, do nothing.
- Otherwise, each recursive call inserts an item from the input list into its correct position in the sorted list so far.

(Wikipedia doesn’t give good specs!)
insertion sort

- If the list is empty, do nothing.
- Otherwise, recursively sort the tail, then insert the head item into its correct position in the sorted tail.

... need a helper function to do insertion

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

\text{REQUIRES} ... 
\text{ENSURES} ...
insertion

ins : int * int list -> int list

REQUIRES L is a sorted list
ENSURES ins(x, L) = a sorted permutation of x::L

fun ins (x, [ ]) = [x]
| ins (x, y::L) = case compare(x, y) of
    |    GREATER => y :: ins(x, L)
    |        _    => x :: y :: L

For all sorted integer lists L,
ins(x, L) = a sorted permutation of x::L.
insertion

ins : int * int list -> int list
REQUIRES   L is a sorted list
ENSURES    ins(x, L) = a sorted perm of x::L

fun ins (x, [ ]) = [x]
| ins (x, y::L) =
   if x>y then y::ins(x, L) else x::y::L

(equivalent code, using if-then-else)
ins equations

For all values $x,y : \text{int}$ and $L : \text{int list}$,

$$\text{ins } (x, [ ]) = [x]$$

$$\text{ins } (x, y::L) =
\quad \text{if } x>y \text{ then } y::\text{ins}(x, L) \text{ else } x::y::L$$

$$\text{ins } (x, y::L) = y::\text{ins}(x, L) \quad \text{if } x>y$$
$$\quad = x::y::L \quad \text{otherwise}$$
**Theorem**
For all sorted integer lists \( L \), all values \( x: \text{int} \),
\[
\text{ins}(x, L) = \text{a sorted permutation of } x::L.
\]

- **Proof**: By induction on *length* of \( L \).
- **Base case**: When \( L \) has length 0, \( L \) is \([\ ]\).
  \( [\ ] \) is sorted, and \( \text{ins}(x, [\ ]) = [x] \) is a sorted perm of \( x::[\ ] \).
- **Inductive case**: Let \( k>0 \) and assume
  
  IH: For all sorted lists \( A \) of length \(< k \), all values \( x \),
  \[
  \text{ins}(x, A) = \text{a sorted perm of } x::A.
  \]
  
  - Let \( L \) be sorted, of length \( k \). Pick \( y, R \) so that \( L=y::R \).
  - \( R \) is sorted, of length \(< k \), and \( y \leq \) all of \( R \).
  - Need to show:
    \[
    \text{ins}(x, y::R) = \text{a sorted perm of } x::(y::R)
    \]
  
inductive case
(some more details)

\[ \text{ins} \ (x, \ y::R) = \begin{cases} y::\text{ins}(x, R) & \text{if } x>y \\ x::y::R & \text{otherwise} \end{cases} \]

- R is sorted, length < k, and \( y \leq \) all of R.
- By IH, \( \text{ins}(x, R) = \) a sorted perm of \( x::R \)
  - If \( x>y \) we have \( \text{ins}(x, y::R) = y::\text{ins}(x,R) \)
    This list is sorted because...
    This list is a perm of \( x::y::R \) because...
  - Otherwise, \( x\leq y \) and \( \text{ins}(x, y::R) = x::y::R \)
    This list is sorted because...
    This list is a perm of \( x::y::R \) because...
- In all cases, \( \text{ins}(x, y::R) = \) a sorted perm of \( x::y::L \)
isort

isort : int list -> int list

REQUIRES   true
ENSURES     isort(L) = a sorted perm of L

fun isort [ ] = [ ]
|    isort (x::L) = ins (x, isort L)

For all values L: int list,
isort L = a sorted permutation of L.
For all values L: int list,
isort L = a sorted permutation of L.

• **Proof**: By induction on length of L.

• **Base case**: for L = [ ].
  Show that isort [ ] = a sorted perm of [ ].

• **Inductive case**: for L = y::R.
  IH: isort R = a sorted perm of R.
  Show: isort(y::R) = a sorted perm of y::R.

Use the *proven* spec for **ins**!
isort
is a total function
from int list to int list

When e evaluates to L,
isort e evaluates to the sorted version of L
a variation

fun isort [ ] = [ ]
| isort (x::L) = ins (x, isort L)

fun isort’ [ ] = [ ]
| isort’ [x] = [x]
| isort’ (x::L) = ins (x, isort’ L)

clause for isort’ [x] seems redundant
variation

isort' : int list -> int list

fun isort' [ ] = [ ]

| isort' [x] = [x]

| isort' (x::L) = ins (x, isort' L)

If in doubt,

test,

then prove
equivalent

- `isort` and `isort'` are extensionally equivalent.
  
  For all `L : int list, isort L = isort' L`.

- Proof? (See lecture notes!)

  No need for extra clause but it doesn’t do any harm
work

• 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 $n > 0$
work

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

  $W_{\text{isort}}(n)$ is $O(n^2)$
Conceptually, a merge sort works as follows:

1. Divide the unsorted list into $n$ sublists, each containing 1 element.

2. Repeatedly Merge sublists to produce new sublists until there is only 1 sublist left.

Wrong! Wrong! Wrong!
Doesn’t say “recursive”...
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)
implementation

- First, design helper functions

  split : int list -> int list * int list

  merge : int list * int list -> int list

(what specs?)
split spec

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.

fun split [ ] = ([ ], [ ])
| split [x] = ([x], [ ])
| split (x::y::L) =

let val (A, B) = split L in (x::A, y::B) end
fun split [] = ([], [])
|    split [x] = ([x], [])
|    split (x::y::L) =
    let val (A, B) = split L in (x::A, y::B) end

the function definition
gives rise to

value equations

that describe
its applicative behavior
split equations

For all values $x, y : int$ and $L : int$ list,

\[
\text{split } [ ] = ([ ], [ ])
\]

\[
\text{split } [x] = ([x], [ ])
\]

\[
\text{split } (x::y::L) = \text{let } val (A, B) = \text{split } L \text{ in } (x::A, y::B) \text{ end}
\]
split equations

For all values $x, y : \text{int}$ and $L : \text{int list}$,

\[
split \[ \] = ([], [])
\]

\[
split [x] = ([x], [])
\]

\[
split (x::y::L) = (x::A, y::B),
\]

where $(A, B) = \text{split } L$

Can be used to calculate split $R$

for any value $R : \text{int list}$

\[
split [4,2,1,3] = ([4,1], [2,3])
\]
split [38, 27, 43, 3, 9, 82, 10]
    = ([38, 43, 9, 10], [27, 3, 82])
Proof: by (strong) induction on length of L

Base cases: L = [ ], [x]
(i) Show that split [ ] = a pair (A, B) such that length(A)\(\approx\)length(B) & A@B is a perm of [ ].
(ii) Show that split [x] = a pair (A, B) such that length(A)\(\approx\)length(B) & A@B is a perm of [x].

Inductive case: L=x::y::R
Induction Hypothesis: split(R) = a pair (A’, B’) such that length(A’)\(\approx\)length(B’) & A’@B’ is a perm of R.
(iii) Show that split(x::y::R) = a pair (A, B) such that length(A)\(\approx\)length(B) & A@B is a perm of x::y::R.

Key facts
split [ ] = ( [ ], [ ])
[ ]@[ ] = [ ]
split [x] = ([x], [ ])
[x]@[ ] = [x]

split (x::y::R) = (x::A’, y::B’)
length(x::A’) \(\approx\) length(y::B’)
(x::A’)(y::B’) is a perm of x::y::R
• We used *strong* induction on length of L rather than simple induction.

• Reason: \texttt{split(x::y::R)} calls \texttt{split(R)} and length of \texttt{R} is *two less* than length of \texttt{x::y::R}. 
• If \( \text{length}(L) > 1 \) and \( \text{split}(L) = (A, B) \), then \( A \) and \( B \) have *smaller* length than \( L \).

• This follows from the spec, using some fairly obvious facts:

  \[
  \begin{align*}
  A@B \text{ is a perm of } L, \text{ so } \\
  \text{length}(A) + \text{length}(B) &= \text{length}(L) \\
  \text{length}(A) \land \text{length}(B) &\text{ differ by 0 or 1} \\
  \text{if } n > 1 \text{ and } n \text{ odd, } (n \div 2) + 1 &< n \\
  \text{if } n > 1 \text{ and } n \text{ even, } n \div 2 &< n
  \end{align*}
  \]
merge

merge : int list * int list -> int list
REQUIRES A and B are sorted lists
ENSURES merge(A, B) = a sorted perm of A@B

fun merge (A, [ ]) = 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)
merge equations

For all values $x, y : \text{int}$ and $A, B : \text{int list}$,

\[
\begin{align*}
\text{merge} \ (A, [ ]) &= A \\
\text{merge} \ ([ ], B) &= B \\
\text{merge} \ (x::A, y::B) &= \text{case} \ \text{compare}(x, y) \ \text{of} \\
& \quad \text{LESS} \Rightarrow x :: \text{merge}(A, y::B) \\
& \quad \text{EQUAL} \Rightarrow x :: y :: \text{merge}(A, B) \\
& \quad \text{GREATER} \Rightarrow y :: \text{merge}(x::A, B)
\end{align*}
\]

Can be used to evaluate $\text{merge}(L,R)$ for all values $L, R : \text{int list}$

\[
\text{merge}([1,4], [2,3]) = [1,2,3,4]
\]
merge equations

For all values $x, y : \text{int}$ and $A, B : \text{int list}$,

merge (A, []) = A
merge ([ ], B) = B
merge (x::A, y::B) = x :: merge(A, y::B) if $x<y$

= x :: y :: merge(A, B) if $x=y$
= y :: merge(x::A, B) if $x>y$

Can be used to evaluate merge(L,R)
for all values $L, R : \text{int list}$

merge([1,4], [2,3]) = [1,2,3,4]
merge ([4, 10, 38, 43], [3, 27, 82])
= [3, 4, 10, 27, 38, 43, 82]
• **Proof**: *strong* induction on *product of lengths* of A, B.

• **Base cases**: (A, [ ]) and ([ ], B).
  (i) Show: if A is sorted, merge(A,[ ]) = a sorted perm of A@[ ].
  (ii) Show: if B is sorted, merge([ ],B) = a sorted perm of [ ]@B.

• **Inductive case**: (x::A, y::B).
  IH: for all pairs(A’, B’) with smaller product of lengths, if A’ & B’ are sorted, merge(A’, B’) = a sorted perm of A’@B’.
  Show: if x::A and y::B are sorted,
  merge(x::A, y::B) = a sorted perm of (x::A)@(y::B).

  (Exercise: fill in the details!)
Does clause order matter here? **NO**

Patterns are

- Exhaustive
- Overlap of first two clauses is harmless

Each yields `merge([], []) = []`

(not true for all function definitions!)

Could use nested **if-then-else** instead of **case**. But we need a 3-way branch, so **case** is better style.
previously

• We defined split and merge
• We proved they meet their specs

\[
\text{split} : \text{int list} \rightarrow \text{int list} \times \text{int list}
\]
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 \( L \).

\[
\text{merge} : \text{int list} \times \text{int list} \rightarrow \text{int list}
\]
REQUIRES A and B are sorted lists
ENSURES merge(\( A, B \)) = a sorted perm of \( A@B \)

Now let’s use them to define
\[
\text{msort} : \text{int list} \rightarrow \text{int list}
\]
msort

msort : int list -> int list

REQUIRES  true
ENSURES msort(L) = a sorted perm of L

fun msort [ ] = [ ]
| 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

[38, 27, 43, 3, 9, 82, 10]

msort

[38, 43, 9, 10] [27, 3, 82]

merge

[9, 10, 38, 43] [3, 27, 82]

[3, 9, 10, 27, 38, 43, 82]
msort equations

For all values $x : \text{int}$ and $L : \text{int list}$,

$$msort \ [\ ] = [\ ]$$

$$msort \ [x] = [x]$$

$$msort \ L = \text{merge}(msort \ A, msort \ B)$$

where $(A, B) = \text{split} \ L$,

if $\text{length} \ L \geq 2$

(where did this side condition come from?)
msort [38, 27, 43, 3, 9, 82, 10]

= merge (msort [38, 43, 4, 10], msort [27, 3, 82])

= merge ([4, 10, 38, 43], [3, 27, 82])

= [3, 4, 10, 27, 38, 43, 82]
proof outline

For all L:int list,
msort(L) = a sorted permutation of L.

• **Method**: by strong induction on *length* of L

• **Base cases**:
  (i) Show msort [ ] = a sorted perm of [ ]
  (ii) Show msort [x] = a sorted perm of [x]

• **Inductive case**: suppose length(L) > 1.
  Inductive hypothesis: for all *shorter* lists R, msort R = a sorted perm of R.
  Show that msort L = a sorted perm of L.
msort

msort : int list -> int list

REQUIRES  true

ENSURES    msort(L) = a sorted perm of L

fun msort [] = []
| msort [x] = [x]
| msort L = let

val (A, B) = split L
val A' = msort A
val B' = msort B

in
merge (A', B')
end
msort

msort : int list -> int list

fun msort [ ] = [ ]
| msort [x] = [x]
| msort L = let
  | val (A, B) = split L
  | in
  |    merge (msort A, msort B)
  | end

is this clause redundant?
fun msort [] = []

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

loops forever
on non-empty lists
the problem

- split \([x] = ([x], [\ ])\)
- msort \([x] \Rightarrow^* (fn \ldots \Rightarrow \ldots) (msort [x], msort \ [\ ])\)

leads to infinite computation
• The **proof** for `msort` relied only on the **specifications** of `split` and `merge`

• Can replace `split` by any other function with the **same specification**, and the same proof would go work!
fun split' [ ] = ([ ], [ ])
| split' [x] = ([ ], [x])
| split' (x::y::L) = let val (A, B) = split' L in (x::A, y::B) end

fun msort' [ ] = [ ]
| msort' [x] = [x]
| msort' L = let
|     val (A, B) = split' L
|   in
|     merge(msort' A, msort' B)
| end;
example

- **split** and **split’** are not **extensionally equivalent**
- But they both satisfy the **specification** used in the correctness proof for **msort** and **msort’**
- ... so **msort** and **msort’** are both correct