today

Sorting an integer list

- specifications and proofs
- asymptotic analysis

SML features
- `datatype` definitions
- boolean connectives
- `case` expressions
- `<>` means ≠
**comparison**

```plaintext
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 $a = b$  \hspace{1cm} (antisymmetric)
  
  If $a \leq b$ and $b \leq c$ then $a \leq c$ \hspace{1cm} (transitive)
  
  Either $a \leq b$ or $b \leq a$ \hspace{1cm} (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 $a = b$ \hspace{1cm} (trichotomy)
A list of integers is \(<\text{sorted}\) (or just \text{sorted})\) if each item in the list is \(\leq\) all later items.

\[
\text{sorted} : \text{int list} \rightarrow \text{bool}
\]

\[
\begin{align*}
\text{fun} & \quad \text{sorted} \; [\; ] = \text{true} \\
\& \quad \text{sorted} \; [x] = \text{true} \\
\& \quad \text{sorted} \; (x::y::L) = \\
& \quad \hspace{1cm} (x \leq y) \quad \text{andalso} \quad \text{sorted}(y::L)
\end{align*}
\]

For all \(L : \text{int list}\),

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

\[
\text{fun sorted [ ]} = \text{true} \\
\text{sorted [x]} = \text{true} \\
\text{sorted (x::y::L)} = \\
\quad \text{(compare(x,y) <> GREATER) andalso sorted(y::L)}
\]

An equivalent definition, using \text{compare}

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 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

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

REQUIRES ... 
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 $L : \text{int list}$,

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

$$\text{ins} \ (x, y::L) = \begin{cases} 
    y::\text{ins}(x, L) & \text{if } x>y \\
    x::y::L & \text{otherwise}
\end{cases}$$

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

$$= x::y::L \text{ otherwise}$$
**Theorem**  
For all sorted integer lists $L$, all values $x$: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 ≤ all of R.
- By IH, \(\text{ins}(x, R) = \text{a sorted perm of } x::R\)
  - 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, \(\text{ins}(x, y::R) = \text{a sorted perm of } x::y::L\)
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.
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 \( L \), \( \text{isort } e \) evaluates to the sorted version of \( L \)
fun isort [] = []
| isort (x::L) = ins (x, isort L)

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

isort' : int list -> int list

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

If in doubt,
test,
then prove
• 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*
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 $n > 0$
• 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”...

… what’s \( n \)?

… repeatedly????

… and then?

What’s the output?
How does it relate to the input?
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

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

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

  (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 \(x, y : \text{int}\) and \(L : \text{int list}\),

\[
\begin{align*}
\text{split } [ ] &= ([ ], [ ])
\end{align*}
\]

\[
\begin{align*}
\text{split } [x] &= ([x], [ ])
\end{align*}
\]

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

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

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

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

\[
\text{split } (x::y::L) = (x::A, y::B),
\text{ where } (A, B) = \text{split } L
\]

Can be used to calculate \(\text{split } R\) for any value \(R : \text{int list}\)

\[
\text{split } [4,2,1,3] = ([4,1], [2,3])
\]
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 \( \text{split} [ \ ] = (A, B) \) such that \( \text{length}(A) \approx \text{length}(B) \) & \( A@B \) is a perm of \( [ \ ] \).
  
  (ii) Show that \( \text{split} [x] = (A, B) \) such that \( \text{length}(A) \approx \text{length}(B) \) & \( A@B \) is a perm of \( [x] \).

• **Inductive case:** \( L = x::y::R \)

  Induction Hypothesis: \( \text{split}(R) = (A', B') \) such that \( \text{length}(A') \approx \text{length}(B') \) & \( A'@B' \) is a perm of \( R \).

  (iii) Show that \( \text{split}(x::y::R) = (A, B) \) such that \( \text{length}(A) \approx \text{length}(B) \) & \( A@B \) is a perm of \( x::y::R \).

  **Key facts**

  \[
  \begin{align*}
  \text{split} [ \ ] &= ([ ], [ ]) \\
  [ ]@[ ] &= [ ] \\
  \text{split} [x] &= ([x], [ ]) \\
  [x]@[ ] &= [x]
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{split} (x::y::R) &= (x::A', y::B') \\
  \text{length}(x::A') &\approx \text{length}(y::B') \\
  (x::A')@(y::B') &\text{ is a perm of } x::y::R
  \end{align*}
  \]
We used *strong* induction on length of $L$ rather than simple induction.

Reason: $\text{split}(x::y::R)$ calls $\text{split}(R)$ and length of $R$ is *two less* than length of $x::y::R$. 
• 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:

$$A@B \text{ is a perm of } L, \text{ so } \text{length}(A) + \text{length}(B) = \text{length}(L)$$

length($A$) & length($B$) differ by 0 or 1

if $n > 1$ and $n$ odd, $(n \text{ div } 2) + 1 < n$

if $n > 1$ and $n$ even, $n \text{ div } 2 < n$
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 $A, B : \text{int list}$,

$$
\begin{align*}
\text{merge} \ (A, [ ]) &= A \\
\text{merge} \ ([ ], B) &= B \\
\text{merge} \ (x::A, y::B) &= \text{case} \ \text{compare}(x, y) \ \text{of} \\
& \hspace{1cm} \text{LESS} \Rightarrow x :: \text{merge}(A, y::B) \\
& \hspace{1cm} | \quad \text{EQUAL} \Rightarrow x :: y :: \text{merge}(A, B) \\
& \hspace{1cm} | \quad \text{GREATER} \Rightarrow y :: \text{merge}(x::A, B)
\end{align*}
$$

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 \( A, B : \text{int list} \),

\[
\begin{align*}
\text{merge} (A, [ ]) &= A \\
\text{merge} ([ ], B) &= B \\
\text{merge} (x::A, y::B) &= x :: \text{merge}(A, y::B) \quad \text{if } x<y \\
&= x :: y :: \text{merge}(A, B) \quad \text{if } x=y \\
&= y :: \text{merge}(x::A, B) \quad \text{if } x>y
\end{align*}
\]

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 ([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? \textbf{NO}

Patterns are \{ Exhaustive, Overlap of first two clauses is harmless \}
Each yields \texttt{merge([ ], [ ]) = [ ]}

(not true for all function definitions!)

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

• We *defined* **split** and **merge**

• We *proved* they meet their specs

\[
\text{split} : \text{int list} \rightarrow \text{int list} * \text{int list}
\]
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.

\[
\text{merge} : \text{int list} * \text{int list} \rightarrow \text{int list}
\]
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]

msort [4,2,1,3] = merge(msort [4,1], msort [2,3])
= merge([1,4], [2,3])
= [1,2,3,4]
split

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

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

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

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

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

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

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

merge
split

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

msort

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

merge

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

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

\[
\text{msort } [ ] = [ ]
\]

\[
\text{msort } [x] = [x]
\]

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

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]
• **Method**: by strong induction on *length* of \( L \)

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

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

an alternative version
fun msort [ ] = [ ]
| msort [x] = [x]
| msort L = let
  val (A, B) = split L
  in
  merge (msort A, msort B)
end
after deletion

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] =>* (fn ... => ...) (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

• \texttt{split} and \texttt{split'} are not \textit{extensionally equivalent}

• But they both satisfy the \textit{specification} used in the correctness proof for \texttt{msort} and \texttt{msort'}

• ... so \texttt{msort} and \texttt{msort'} are both correct