Most of the time I don't have much fun. The rest of the time I don't have any fun at all.
today

Sorting an integer list

• Using specifications to guide program design

• “Helper functions should help!”

SML features

• `datatype` definitions
• `boolean` connectives
• `case` expressions
• `<>` means ≠
datatypes

- ML has datatype declarations
- Allow us to introduce new types, with constructors for building values

```haskell
datatype order = LESS | EQUAL | GREATER

datatype 'a option = NONE | SOME of 'a

NONE : int option
SOME 42 : int option
```

'a list is a built-in datatype with constructors nil and ::
comparing ints

datatype order = LESS | EQUAL | GREATER
comparing ints

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
comparing ints

**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** : int * int -> order

compare(x,y) = LESS if x<y
compare(x,y) = EQUAL if x=y
compare(x,y) = GREATER if x>y
properties of $<$ and $\leq$ on integers

• $\leq$ is a **linear ordering**

  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} (connected)

• $<$ 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 is `<-sorted` (or just `sorted`) if and only if each item in the list is ≤ all later items.

```
fun sorted [] = true
|    sorted [x] = true
|    sorted (x::y::L) = (x <= y) andalso sorted(y::L)
```
A list is <\textit{sorted} \textit{(or just sorted)} if and only if each item in the list is $\leq$ all later items.

\begin{verbatim}
fun sorted [ ] = true
  | sorted [x] = true
  | sorted (x::y::L) = (x <= y) andalso sorted(y::L)
\end{verbatim}

For all L : int list,

\begin{align*}
  \text{sorted}(L) &= \text{true} \quad \text{if L is sorted} \\
  &= \text{false} \quad \text{otherwise}
\end{align*}
sorted

sorted : int list -> bool

A list is \(-sorted\) (or just \(sorted\)) if and only if each item in the list is \(\leq\) all later items.

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

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

\[
\text{sorted}(L) = \text{true} \quad \text{if } L \text{ is } sorted \\
= \text{false} \quad \text{otherwise}
\]

(Prove this, by induction on list length)
(Note the relevance of transitivity etc.)
specs and code

• We use sorted only in specifications.
• Our sorting functions won’t use it.
• But you could use it for testing...
specs and code

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

For every integer list $L$ *there is a unique sorted permutation of* $L$.
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.
**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 (already sorted) tail.
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 (already sorted) tail.

  … We need a helper function
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 (already sorted) tail.

... We need a helper function

```plaintext
ins : int * int list -> 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

inserts x into its correct position in L

permutation

noun
A way, in which a list of things can be arranged:
"his thoughts raced ahead to fifty different permutations of what he must do"
Powered by Oxford Dictionaries
**insertion**

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

**REQUIRES** \( \text{L is a sorted list} \)

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

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

**permutation**

*noun*

A way, in which a list of things can be arranged:

"his thoughts raced ahead to fifty different permutations of what he must do"
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]
l   ins (x, y::R) =
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::R) = if x > y

per·mu·ta·tion

noun
A way, in which a list of things can be arranged:
"his thoughts raced ahead to fifty different permutations of what he must do"

Powered by Oxford Dictionaries
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::R) = if x > y
| then y :: ins(x, R)

per·mu·ta·tion
noun
A way, in which a list of things can be arranged:
"his thoughts raced ahead to fifty different permutations of what he must do"
Powered by Oxford Dictionaries
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::R) = if x > y
|        then y :: ins(x, R)
|    else   x :: (y :: R)

insertion

per·mu·ta·tion
noun
A way, in which a list of things can be arranged:
"his thoughts raced ahead to fifty different permutations of what he must do"
Powered by Oxford Dictionaries
ins equations

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

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

\[
\text{ins} \ (x, y::R) = \\
\quad \text{if } x > y \text{ then } y::\text{ins}(x, R) \text{ else } x::(y::R)
\]

\[
\text{ins} \ (x, y::R) = y::\text{ins}(x, R) \quad \text{if } x > y
\]
\[
\quad = x::(y::R) \quad \text{otherwise}
\]
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 \( L \) be sorted, of length \( k \).
  IH: For all sorted lists \( A \) of length < \( k \), all values \( x \),
  \( \text{ins}(x, A) = \text{a sorted perm of } x::A \).
  Let \( y, R \) be the head, tail of \( L \): so \( L = y::R \).
  \( R \) 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)

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

- \( R \) 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 \( x::(y::R) \)
Fill in the missing details in that proof sketch

Notice where you use basic properties of $\leq$

- these properties are crucial
- often used implicitly, without mention
- that’s OK, except that you need to realize it

Now that we have ins, let’s define isort…
isort

isort : int list -> int list

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

isort : int list -> int list

ENSURES isort(L) = a sorted perm of L

fun isort [ ] = [ ]
isort

isort : int list -> int list

ENSURES isort(L) = a sorted perm of L

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

isort : int list -> int list

ENSURES isort(L) = a sorted perm of L

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

“isort (x::R) inserts x into its correct position in the sorted tail, isort R”
Proof: By \textit{structural induction} on $L$.

- **Base case**: for $L = [ ]$.
  Show that $\text{isort } [ ] = \text{a sorted perm of } [ ]$.

- **Inductive case**: for $L = y::R$.
  IH: $\text{isort } R = \text{a sorted perm of } R$.
  Show: $\text{isort}(y::R) = \text{a sorted perm of } y::R$.

\[
\text{isort } (y::R) = \text{ins } (y, \text{isort } R) \\
\text{isort } R \text{ is a sorted perm of } R \\
\text{By the proven ins spec, it follows that} \\
\text{ins } (y, \text{isort } R) = \text{a sorted perm of } y::R
\]
• The proof was “by **structural induction** on \( L \)”
  • Every list value \( L \) is either \([\ ]\) (nil) or \( y::R \), where \( R \) is a “smaller” list value

• We could just as well have said “by **induction on length** of \( L \)”
  • \([\ ]\) has length 0
  • \( 0 \leq \text{length } R < \text{length}(y::R) \)
perm facts

\[ y::\text{(a perm of } R\text{)} \] is a perm of \( y::R \)

A perm of a perm of \( L \) is a perm of \( L \)

In the correctness proof we used some obvious facts about permutations.
isort is a total function from int list to int list
corollaries

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 $L$
a variation

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

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

| isort (x::R) = ins (x, isort R)

fun isort' [ ] = [ ]

| isort' [x] = [x]

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

is this clause redundant
variation

\texttt{isort'} : \textit{int list} -> \textit{int list}

\texttt{fun \: isort'} \: [ ] = [ ]
\texttt{\quad | \quad isort'} \: [x] = [x]
\texttt{\quad | \quad isort'} \: (x::R) = \text{ins} \: (x, \text{isort'} \: R)

\text{If in doubt,}
\text{\hspace{1cm} test,}
\text{\hspace{1cm} then prove}
**variation**

`isort' : int list -> int list`

```plaintext
fun isort' [ ] = [ ]
| isort' [x] = [x]
| isort' (x::R) = ins (x, isort' R)
```
variation

isort' : int list -> int list

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

If in doubt,
    test,
then prove
equivalent

• *isort* and *isort’* are extensionally equivalent:

  For all \( L : \text{int list} \), \( \text{isort} \ L = \text{isort’} \ L \).

• Proof? See lecture notes…

  OR: Re-do the *isort* proof for *isort’* (easy)

  Hence they satisfy the same spec, so

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

  \( \text{isort} \ L = \text{isort’} \ L = \text{the sorted perm of } L \)
equivalent

• `isort` and `isort'` are extensionally equivalent:
  
  For all \( L : \text{int list} \), \( \text{isort} \ L = \text{isort'} \ L \).

• Proof? See lecture notes…

  OR: Re-do the `isort` proof for `isort'` (easy)

  Hence they satisfy the same spec, so
  
  For all \( L : \text{int list} \),
  
  \( \text{isort} \ L = \text{isort'} \ L = \text{the sorted perm of} \ L \)

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 $n$.

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

- Let $W_{ins}(n)$ be the work for $\text{ins}(x, L)$ when $x, L$ are values and $L$ has length $n$
  
  $W_{ins}(n)$ is $O(n)$

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

  $W_{isort}(0) = 1$
  
  $W_{isort}(n) = 1 + W_{ins}(n-1) + W_{isort}(n-1)$
  
  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 $n$
• 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 $n > 0$
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 $O(n^2)$
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 $O(n^2)$

This is slow! We can do better!
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.
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!
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”...
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”...
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”...
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”...
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.
- Otherwise,

  *split* the list into two shorter lists,
  *sort* these two lists,
  *merge* the (sorted) results
implementation

• First, let’s design helper functions

  \texttt{split : int list \to int list \times int list}

  \texttt{merge : int list \times int list \to int list}
implementation

- First, let’s design helper functions
  
  split : int list -> int list * int list
  merge : int list * int list -> int list

  (what specs should we use?)
implementation

- First, let’s design helper functions

  split : int list -> int list * int list
  merge : int list * int list -> int list

  (what specs should we use?)

  - split splits a list into two sublists
  - merge combines two sorted lists into one
implementation

- First, let’s design helper functions

  split : int list -> int list * int list
  merge : int list * int list -> int list

  (what specs should we use?)

  split  splits a list into two sublists
  merge  combines two sorted lists into one

  (a bit imprecise, but we’ll fix that…)

split

split : int list -> int list * 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.
split

split : int list -> int list * 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.

write as
length(A) \approx length(B)
split

split : int list -> int list * 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.
split

split : int list -> int list * 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.

fun split [ ] = ([ ], [ ])
split

split : int list -> int list * 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.

fun split [ ] = ([ ], [ ])
| split [x] = ([x], [ ])
split

split : int list -> int list * 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.

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

split : int list -> int list * 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.

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

split : int list -> int list * 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.

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

    let val (A, B) = split L in (x::A, y::B) end
split

split : int list -> int list * 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.

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

note the use of
list patterns and pair patterns
split equations

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

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

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

\[
\text{split } (x :: y :: L) =
\]

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

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

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

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

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

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

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

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

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

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

Can be used to calculate $\text{split } R$

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

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

\[
\begin{align*}
\text{split } [ ] &= ([ ], [ ]) \\
\text{split } [x] &= ([x], [ ]) \\
\text{split } (x::y::L) &= (x::A, y::B), \\
&\quad \text{where } (A, B) = \text{split } L
\end{align*}
\]

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 equations

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

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

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

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

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

\[
\text{split } [4,2,1,3] = ([4,1], [2,3])
\]
\[
\text{split } [4,2,1] = ([4,1], [2])
\]
For all L:int list,  
\[ \text{split}(L) = \text{a pair of lists } (A, B) \text{ such that } \text{length}(A) \approx \text{length}(B) \text{ and } A@B \text{ is a permutation of } L. \]

- **Proof**: by (strong) induction on *length* of \( L \)

- **Base cases**: \( L = [\ ], [x] \)  
  EASY

- **Inductive case**: \( L = x::(y::R) \)  
  \( R \) is shorter than \( L \)  
  **Assume** Induction Hypothesis: \( \text{split}(R) = \text{a pair } (A', B') \text{ such that } \text{length}(A') \approx \text{length}(B') \text{ and } A'@B' \text{ is a perm of } R. \)  
  **Show** that \( \text{split}(x::y::R) = \text{a pair } (A, B) \text{ such that } \text{length}(A) \approx \text{length}(B) \text{ and } A@B \text{ is a perm of } x::(y::R). \)
• **Proof:** by (strong) induction on *length* of \( L \)

• **Base cases:** \( L = [\ ], [x] \)
  
  EASY

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

• **Inductive case:** \( L = x::(y::R) \quad \text{\( R \) is shorter than \( L \)} \)

  **Assume** Induction Hypothesis: \( \text{split}(R) = \) a pair \((A’, B’)\) such that \( \text{length}(A’) \approx \text{length}(B’) \) and \( A’@B’ \) is a perm of \( R \).

  **Show** that \( \text{split}(x::y::R) = \) a pair \((A, B)\) such that \( \text{length}(A) \approx \text{length}(B) \) and \( A@B \) is a perm of \( x::(y::R) \).
• **Proof**: by (strong) induction on *length* of L

• **Base cases**: \( L = [\ ], [x] \)
  
  EASY

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

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

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

\[
\text{split}(x::y::R) = (x::A', y::B') \\
\text{length}(x::A') \approx \text{length}(y::B') \quad (x::A')@(y::B') \text{ is a perm of } x::(y::R)
\]
• We used strong induction on length of L

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

• If length \( L = n > 1 \) and \( \text{split}(L) = (A, B) \), A and B are shorter than L

  If \( n \) is even > 1, length \( A = \text{length } B = n \text{ div } 2 < n \).
  If \( n \) is odd > 1, length \( A = (n \text{ div } 2) + 1 < n \), length \( B = n \text{ div } 2 < n \).
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
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

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::L, y::R) = case compare(x, y) of
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::L, y::R) = case compare(x, y) of
                       LESS  => x :: merge(L, y::R)
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::L, y::R) = case compare(x, y) of
  LESS  => x :: merge(L, y::R)
  |   EQUAL  => x :: y :: merge(L, R)

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::L, y::R) = case compare(x, y) of
    LESS  => x :: merge(L, y::R)
    | EQUAL  => x :: y :: merge(L, R)
    | GREATER => y :: merge(x::L, R)
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::L, y::R) = case compare(x, y) of
|     LESS  => x :: merge(L, y::R)
|     EQUAL => x :: y :: merge(L, R)
|     GREATER => y :: merge(x::L, R)

We need a 3-way branch,
so cased comparison is better than nested if-then-else
merge equations

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

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

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

\[
\begin{align*}
\text{merge } (A, [ ]) &= A \\
\text{merge } ([ ], B) &= B
\end{align*}
\]
merge equations

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

merge $(A, [\ ]) = A$
merge $([\ ], B) = B$
merge $(x::A, y::B) =$
  $x :: \text{merge}(A, y::B) \quad \text{if } x<y$
  $= x :: y :: \text{merge}(A, B) \quad \text{if } x=y$
  $= y :: \text{merge}(x::A, B) \quad \text{if } x>y$
merge equations

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

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

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

$$
\text{merge}([1,4], [2,3]) = [1,2,3,4]
$$
How do we prove this function satisfies the spec?

- Induction, but on on what?
  - in base cases, at least one list is empty
  - in recursive calls, one or both is shorter
correctness?

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

How do we prove this function satisfies the spec?

• Induction, but on on what? The product of list lengths!
  - in base cases, at least one list is empty
  - in recursive calls, one or both is shorter
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)
  Assume IH: for all pairs of sorted lists (A’, B’) with smaller product of lengths than (x::A, y::B), merge(A’, B’) = a sorted perm of A’@B’.
  Show: if x::A and y::B are sorted, then merge(x::A, y::B) = a sorted perm of (x::A)@(y::B).

Exercise: fill in the details!
**msort**

- We proved that **split** and **merge** are correct

**split** : int list -> int list * int list

ENSURES split L = a pair of lists (A, B) such that length(A) ≈ length(B) and A@B is a perm of L

**merge** : int list * int list -> 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 **msort**

**msort** : int list -> int list

ENSURES msort L = a sorted perm of L
msort

msort : int list -> int list

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

msort : int list -> int list

ENSURES msort(L) = a sorted perm of L

fun msort [ ] = [ ]
msort

msort : int list -> int list

ENSURES msort(L) = a sorted perm of L

fun msort [ ] = [ ]
| msort [x] = [x]
msort

msort : int list -> int list

ENSURES  msort(L) = a sorted perm of L

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

msort : int list -> int list

ENSURES msort(L) = a sorted perm of L

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

msort : int list -> int list

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

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

msort : int list -> int list

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**

`msort : int list -> int list`

ENSURES  \( \text{msort}(L) = \text{a sorted perm of } L \)

```ml
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]
```

```ml
msort [4,2,1,3]
  = merge(msort [4,1], msort [2,3])
  = merge([1,4], [2,3])
  = [1,2,3,4]
```
[38, 27, 43, 3, 9, 82, 10]
msort equations

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

\[
\begin{align*}
\text{msort } [ ] &= [ ] \\
\text{msort } [x] &= [x] \\
\text{msort } L &= \text{merge(\text{msort } A, \text{msort } B)} \\
&\text{where } (A, B) = \text{split } L, \\
&\text{if } \text{length } L \geq 2 \\
&\text{(where did this side condition come from?)}
\end{align*}
\]
correctness

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

Proof: 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.

A crucial assumption needed in proof details: length L > 1
• The \texttt{msort} proof was
  “by (strong) induction on the length of \textit{L}”
  
  - \texttt{msort L} calls \texttt{msort A} and \texttt{msort B},
    where \texttt{A} and \texttt{B} have shorter length

• It would not have been appropriate to say
  “by structural induction on \textit{L}”

  - \texttt{msort (x::R)} doesn’t call \texttt{msort R}
work

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

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

\( W_{\text{merge}}(n) = \text{work of merge(A, B)} \)

when \( \text{length(A) + length(B) = n} \)

\( W_{\text{merge}}(n) \text{ is } O(n) \)
work

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

\[ W_{msort}(0) = 1 \]
\[ W_{msort}(1) = 1 \]

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

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

\[ W_{msort}(n) \text{ is } O(n \log n) \]
Deriving the work for msort

Simplify recurrence to:

\[ W(n) = n + 2 W(n \text{ div } 2) \]
\[ = n + 2 \left( n \text{ div } 2 + 2 W(n \text{ div } 4) \right) \]
\[ = n + 2(n/2) + 4 W(n/4) \]
\[ = n + 2(n/2) + 4(n/4) + 8 W(n/8) \]
\[ = n + 2(n/2) + 4(n/4) + \ldots + 2^k (n/2^k) \]

where \( k = \log_2 n \)

\[ = n + n + n + \ldots + n \quad (k \text{ terms}) \]
\[ = n \log_2 n \]

This \( W \) has same asymptotic behavior as \( W_{\text{msort}} \)

So \( W_{\text{msort}}(n) \) is \( O(n \log n) \)
summary

- `msort` is correct
  
  \[ msort \, L = isort \, L = \text{sorted perm of } L \]

- `msort` is (more) efficient (than `isort`)
  
  \[ W_{msort}(n) \text{ is } O(n \log n) \]
  
  \[ W_{isort}(n) \text{ is } O(n^2) \]
Variations on a Theme by W. A. Mozart

from "The Magic Flute"

Andante Largo, Op. 9

F. S.
Let’s consider some alternative ways to write this function.

Some will be correct, some not.
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
✓ correct
✓ work
fun msort L = 
  if length L < 2 
  then L 
  else let 
    val (A, B) = split L 
    in 
      merge (A, B) 
    end
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

is this clause redundant?
fun msort [ ] = [ ]
  | msort L = let
  |   val (A, B) = split L
  |   in
  |     merge (msort A, msort B)
  |   end
fun msort [ ] = [ ]

| msort L = let

val (A, B) = split L

in

merge (msort A, msort B)
end
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 \([x] = ([x], [])\)
- \(\text{msort} [x] \xrightarrow{\ast} (\text{fn} \ldots \Rightarrow \ldots) (\text{msort} [x], \text{msort} [\ ]))\)

leads to infinite computation
the problem

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

- \(\text{msort } [x] \rightarrow^* (\text{fn } \ldots \Rightarrow \ldots) (\text{msort } [x], \text{msort } [])\)

leads to infinite computation

Q: What happens if you try to prove \text{msort} correct?
the problem

- split \([x] = ([x], [])\)
- \(\text{msort} \ [x] \longrightarrow^* (\text{fn} \ ... \Rightarrow \ ...) \ (\text{msort} \ [x], \text{msort} \ [\ ]))\)

leads to infinite computation

Q: What happens if you try to prove \(\text{msort}\) correct?

A: The proof breaks down!
the problem

- split \([x] = ([x], [\ ])\)
- \(\text{msort } [x] \implies^* \text{(fn } ... \Rightarrow ... \text{)} \text{(msort } [x], \text{msort } [\ ])\)

leads to infinite computation

Q: What happens if you try to prove \text{msort} correct?

A: The proof breaks down!

Cannot assume length \(L > 1\) in inductive step
THE JOY of SPEX
• 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 work!
• 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 work!

**THANKS CAPTAIN OBVIOUS**

*the new version of *msort* also sorts lists!*
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
• **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
clause order

fun merge (A, [ ]) = A
|    merge ([ ], B) = B
|   merge (x::A, y::B) = ...

or

fun merge (x::A, y::B) = ...
|    merge (A, [ ]) = A
|   merge ([ ], B) = B

ML tries patterns in the order written
ML tries patterns in the order written

Does clause order matter here?
ML tries patterns in the order written

Does clause order matter here? **NO**
Does clause order matter here? **NO**

Patterns are *exhaustive*

Overlap of first two clauses is *harmless*

Each yields \( \text{merge}([ ], [ ]) = [ ] \)
• The helper functions really helped
• They were also useful for testing
• But we only really cared about \texttt{isort, msort}

\begin{verbatim}
Standard ML of New Jersey
- \textbf{fun} ins ... \\
- \textbf{fun} isort ... ; \\
val ins = fn : int * int list -> int list \\
val isort = fn : int list -> int list
\end{verbatim}

\begin{verbatim}
Standard ML of New Jersey
- \textbf{fun} split ... \\
- \textbf{fun} merge ... \\
- \textbf{fun} msort ... ; \\
val split = fn : int list -> int list * int list \\
val merge = fn : int list * int list -> int list \\
val msort = fn : int list -> int list
\end{verbatim}
There may be no good reason to make the helper functions *visible* to the entire world

We can easily make them “private”

---

Standard ML of New Jersey

```ml
- local
  fun ins ... 
  in 
    fun isort ... 
  end;
val isort = fn - : int list -> int list
```

```
- local
  fun split ...
  fun merge ...
  in 
    fun msort ...
  end;
val msort = fn - : int list -> int list
```
conclusion

• We implemented two well known sorting algorithms for integer lists
  • insertion sort
  • mergesort
• There are many others…
  • quicksort
  • bubble sort
  • selection sort
conclusion

• We implemented two well known sorting algorithms for integer lists
  • insertion sort
  • mergesort
coming soon

• Generalizing from \texttt{int} to an \textit{ordered type}

• Generalizing from \textit{lists} to \textit{trees} of data

Advantages of
\textit{functional programming}