15-150 Fall 2023

Lecture 6
Sorting lists of integers

1 Outline

In this lecture we put together the themes introduced so far

- recursive function definition
- inductive proof techniques
- specifications, correctness, and efficiency

and we develop functional programs that implement the insertion sort and mergesort algorithms for sorting lists of integers. You should be able to do similarly with other well known sorting algorithms, such as quicksort. Later we will generalize to sorting lists of other data, with respect to different orderings; and we'll talk about sorting trees and other structured data. We'll even develop some *parallel* sorting functions.

For now, it would help to review your knowledge of basic properties of the integers and the "usual" $<$ ordering. If you are not familiar with the notation and basic facts you should consult a discrete math text or look online! You have probably been using many of these properties without much thought, but here we will need to be explicit so that you see their relevance in proving correctness of programs. In particular, for each pair of integers m and n exactly one of the following three relationships holds: either $m < n$, or $n < m$, or $m = n$. We write $x \leq y$ for $x < y$ or $x = y$, and we call $\leq$ the less-than-or-equal-to relation. It satisfies the "transitive" property: $x \leq y$ & $y \leq z$ implies $x \leq z$. It is also *anti-symmetric*, in that $x \leq y$ & $y \leq x$ implies $x = y$. Note also that for any pair of integers we always have either $x \leq y$ or $y \leq x$; Wikipedia calls this the *connex property*, but I am unfamiliar with that terminology! I prefer to call it "totality" of the $\leq$ relation. In summary, this combination of properties (anti-symmetry, transitivity, and totality) characterizes what is known as a *linear* (or *total*) ordering. So what we're saying is that the usual less-than-or-equal relation on the integers is a total ordering.

It's often more convenient to refer to $<$ (the strictly less-than relation) instead of $\leq$, but the two notions are intimately connected so there shouldn't be any confusion. Of course, the $<$ relation on the integers also satisfies transitivity: $x < y$ and $y < z$ implies $x < z$.

And it's common to use the word "order" instead of "ordering", and I reserve the right to use these terms interchangeably since I've been doing that for years now!

2 Remarks

In the code below we provide accurate specifications but we don't always use the REQUIRES and ENSURES format. In Lecture 5 we discussed much of this code. These notes supplement what was said and done in class, offering a different perspective and often more explicit detail. Some of the discussion of *work* in these notes may occur in Lecture 6, depending on time.

3 Background

Comparison in ML

We introduce the following type and function; each is already pre-defined in SML, but we give explicit definitions since we use them here for the first time. (Actually the function `compare` is called `Int.compare` in SML. We give it a shorter name just for convenience.)

```
datatype order = LESS | EQUAL | GREATER

(* A comparison function for integers *)
(* compare : int * int -> order       *)

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 *)
```

The first line above is a simple form of *datatype* definition. It introduces a user-defined type named `order`. Later in the semester we will see more sophisticated datatype definitions. The ability to define your own types like this, and have them fit in seamlessly with the rest of the ML type discipline, is a very powerful feature of the programming language.

The type `order` has just three (syntactic) values, written `LESS`, `EQUAL`, and `GREATER`, each in all-caps. You can test expressions of type `order` for equality (using ML `=`) or use one in a `case`-expression to do different things based on a comparison result. (The type `order` is an *equality type*.) ML also allows the use of `<>` ("not equal to") on values of an equality type.

By the way, the infix `<` operator in ML, of type `int * int -> bool`, behaves just like the "usual" less-than ordering on the integers in math. In math we write $x \leq y$ for "x is less-than-or-equal-to y", but in ML we use `x <= y`. Of course `int` is also an equality type, so we can use ML `=` to test for equality of integer expressions.

Linear ordering

The $\leq$ relation on integers has some well known properties. These are crucial in the design and analysis of sorting algorithms. In particular, $\leq$ is a *linear ordering*. This means that for all integers x, y, z:

- $x \leq y$ & $y \leq x$ implies $x = y$ (antisymmetry)
- $x \leq y$ & $y \leq z$ implies $x \leq z$ (transitivity)
- $x \leq y$ or $y \leq x$ (totality)

The property called "totality" is also known as "connectivity".

Sorted lists

A list of integers is *sorted* if each item in the list is $\leq$ all items that occur later in the list. Here is an ML function that checks for this property.
We only use this function in specifications!

```
(* sorted : int list -> bool *)
fun sorted [ ] = true
 |  sorted [x] = true
 |  sorted (x::y::L) =
      (compare(x,y) <> GREATER) andalso sorted(y::L)

(* REQUIRES L is an integer list                 *)
(* ENSURES sorted L = true iff L is a sorted list *)
```

The expression `compare(x,y) <> GREATER` evaluates to `true` if the value of `compare(x, y)` is `LESS` or `EQUAL`, and evaluates to `false` if `x` $>$ `y`.

Examples:

```
sorted [1,2,3] = true
sorted [3,2,1] = false
```

Make sure you notice the relevance of the *linear ordering properties* here: they are the reason why this `sorted` function behaves as described! They also justify why it's not necessary in the third clause of the function to check that `x` is not greater than *all* the elements of `L`; the term `sorted(y::L)` checks that `y` is less-than-or-equal to the elements of `L`, and the knowledge that `x` $\leq$ `y` is enough (because of *transitivity*) to then imply that `x` is less-than-or-equal to all items in `L`.

From now on we will say that an integer list `L` is *sorted* if and only if `sorted(L)` evaluates to `true`. We will assume you remember the basic properties described above, and we will take advantage of "obvious" properties, such as: an integer list of length 1 is sorted, and an integer list of form $[x, y]$ is sorted if and only if $x \leq y$. Usually any facts that we deem "obvious" are easy to prove from the definition of sortedness, and rely on the basic properties of linear orderings.

4 Insertion sort

Here is a function that implements the well known *insertion sort* algorithm. This algorithm is often (informally) described as building up its sorted output by starting with an empty list and successively inserting the items from the input list, at each stage maintaining the correct sorted order. Unfortunately this kind of description is a bit too too vague to be precise, using fuzzy words like "building up" and "successively". A better (more precise) description would be:

> To *insertion sort* a list: if the list is empty, do nothing; otherwise, (recursively) *insertion sort* the tail of the list and then *insert* the head.

(This version is clearer about order of evaluation.) Of course we need a helper function for *inserting* an item into a list, and we can see from the above that we will only need to do insertion *into* an already sorted list, and we need maintain sortedness when we do so!

Let's code this algorithm in SML. We won't need to use ML functions for extracting heads and tails of lists; instead we'll use pattern matching.

First we define a helper function for inserting an integer into its proper place in a sorted list. To be clear about what this means, we refer to the familiar notion of *permutation.* An integer list `A` is a permutation of a list `B` if `A` contains the same items as `B`, possibly in a different order, and each integer occurs the same number of times in `A` as in `B`. For example, `[1,2,3,1]` is a permutation of `[2,1,3,1]` and a permutation of `1::(1::2::[3])`, but not a permutation of `[1,2,3]`.

```
(* ins : int * int list -> int list *)
(* REQUIRES  L is a sorted list of integers            *)
(* 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
```

Examples:

```
   ins (2, [1,3]) = [1,2,3]
   ins (2, [3,1]) = [2,3,1]
   ins (2, [1,2,3]) = [1,2,2,3]
```

As the ins function does a three-way comparison but does the same thing in two outcomes it would be just as acceptable stylistically to use an if-then-else with a single $<$ test:

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

Using `ins` as a helper, we can implement insertion sort as follows:

```
(* isort : int list -> int list *)
(* REQUIRES L is an integer list                *)
(* ENSURES isort(L) = a sorted permutation of L *)

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

In the lecture slides we sketched a proof that `ins` satisfies its specification, and that `isort` satisfies its specification. Be sure to study these proofs (and fill in any missing details). They are a very useful exercise in understanding how recursion and induction work.

Efficiency analysis for `isort`

It's easy to see that the work to evaluate `ins(x, L)` when `x` and `L` are values of type `int` and `int list` depends on the *length* of `L` (and on the results of the comparisons between `x` and the items in `L`). We want to assume the worst, to get an upper bound on evaluation time. It's easy to see that the worst case actually happens when `x` is greater than all items in `L`. Let $\mathtt{W}_{\mathtt{ins}}(n)$ be the (worst-case) work to evaluate `ins(x, L)` when `L` ranges over lists of length `L`. We can extract the following recurrence from the definition of `ins` (and simplifying by choosing constants to be 1):

$$\begin{aligned} \mathtt{W}_{\mathtt{ins}}(0) &= 1 \\ \mathtt{W}_{\mathtt{ins}}(n) &= 1 + \mathtt{W}_{\mathtt{ins}}(n-1) \qquad \text{for } n > 0 \end{aligned}$$

Clearly this implies that $\mathtt{W}_{\mathtt{ins}}(n)$ is $\mathtt{O}(n)$. Similarly the work for `isort L` when `L` is a value depends on the length of `L`, and we can derive the following recurrence for $\mathtt{W}_{\mathtt{isort}}(n)$, the worst case work for `isort L` as `L` ranges over lists of length n:

$$\begin{aligned} \mathtt{W}_{\mathtt{isort}}(0) &= 1 \\ \mathtt{W}_{\mathtt{isort}}(n) &= \mathtt{W}_{\mathtt{ins}}(n-1) + \mathtt{W}_{\mathtt{isort}}(n-1) \qquad \text{for } n > 0 \\ &= \mathtt{O}(n) + \mathtt{W}_{\mathtt{isort}}(n-1) \end{aligned}$$

and this tells us that $\mathtt{W}_{\mathtt{isort}}(n)$ is $\mathtt{O}(n^2)$.

To summarize: `isort` is a *correct* sorting function for integer lists (because we proved that it satisfies the sorting spec); but its running time (for list values of length n) is quadratic, i.e. proportional to n^2. As you must know by now, there are other sorting algorithms with better efficiency. We'll discuss some soon. But first there's still some mileage to be squeezed out of insertion sort.

A variation

Just for interest, here is a slight variation on the insertion sort theme:

```
(* isort2 : int list -> int list *)
fun isort2 [ ] = [ ]
 |  isort2 [x] = [x]
 |  isort2 (x::L) = ins (x, isort2 L)
```

It looks the same except that we've written in an explicit clause for singleton lists (and changed the function's name so we can distinguish it from the original `isort`). We can show easily that this `isort2` function is extensionally equivalent to `isort` as defined above. (So the singleton clause is irrelevant.) Indeed, it is very straightforward to prove, by induction on n, that:

For all $n \geq 0$ and all integer values $x_1, \ldots, x_n$,

$$\texttt{isort}\ [x_1, \ldots, x_n] = \texttt{isort2}\ [x_1, \ldots, x_n].$$

- For $n = 0$, we have $\texttt{isort}\ [\,] = [\,] = \texttt{isort2}\ [\,]$.
- For $n = 1$, we have

$$\begin{array}{rcll} \texttt{isort}\ [x_1] & = & \texttt{ins}(x_1,\ \texttt{isort}\ [\,]) & \text{by def of } \texttt{isort} \\ & = & \texttt{ins}(x_1,\ [\,]) & \text{by def of } \texttt{isort} \\ & = & [x_1] & \text{by def of } \texttt{ins} \\ & = & \texttt{isort2}\ [x_1] & \text{by def of } \texttt{isort2} \end{array}$$

- For the inductive step, suppose $n > 1$ and the property holds for all lists of length $n - 1$, i.e.

  (IH): For all integer values $y_1, \ldots, y_{n-1}$,

$$\texttt{isort}\ [y_1, \ldots, y_{n-1}] = \texttt{isort2}\ [y_1, \ldots, y_{n-1}].$$

  Let `L` be an integer list of length n, of the form $[x_1, \ldots, x_n]$. By the function definitions we have

$$\begin{array}{rcll} \texttt{isort}\ [x_1, \ldots, x_n] & = & \texttt{ins}(x_1,\ \texttt{isort}\ [x_2, \ldots, x_n]) & \text{by def of } \texttt{isort} \\ & = & \texttt{ins}(x_1,\ \texttt{isort2}\ [x_2, \ldots, x_n]) & \text{by IH} \\ & = & \texttt{isort2}\ [x_1, x_2, \ldots, x_n]) & \text{by def of } \texttt{isort2} \end{array}$$

Permutations

Here are some SML functions to help understand permutations:

```
   mem : int * int list -> bool
   del : int * int list -> int list
   perm : int list * int list -> bool

   fun mem(x:int, [ ]) = false
    |  mem(x, y::L) = (x=y) orelse mem(x,L)
(* ENSURES mem(x,L) = true if x occurs in L, false otherwise *)

   fun del(x, y::R) = if x=y then R else y::del(x,R)
(* REQUIRES x occurs in L *)
(* ENSURES del(x,L) = a list containing all items in L except for
                          the first occurrence of x *)

   fun perm([ ], [ ]) = true
    |  perm(_::_, [ ]) = false
    |  perm([ ], _) = false
    |  perm(x::L, R) = mem(x, R) andalso perm(L, del(x,R))
(* ENSURES perm(L,R) = true if L is a permutation of R, false otherwise *)
```

We say that `L` is a permutation of `R` if the two lists have the same elements (the same number of times, for each one) but possibly in a different relative order. This is the case if and only if `perm(L, R) = true`.

Some pretty obvious facts:

- The only permutation of `[ ]` is `[ ]`. Every list is a permutation of itself.
- $x :: P$ is a permutation of L if and only if x is a member of L and P is a permutation of $\texttt{del}(x, L)$.
- If P is a permutation of L, then $x :: P$ is a permutation of $x :: L$.
- `(x::A)@(y::B)` is a permutation of `x::y::(A@B)`
- If P is a permutation of L, then $\texttt{length}\ P = \texttt{length}\ L$.
- If P is a permutation of L and Q is a permutation of P, then Q is a permutation of L. (Loosely speaking, "A permutation of a permutation is a permutation.")

We allow use of these facts without proof.

5 Mergesort

As is well known, mergesort is an algorithm that sorts lists of n integers in time O(n *log* n). Let's implement the algorithm in ML and prove its correctness, then confirm the work estimate.

To mergesort a list of integers, if it is empty or a singleton do nothing (it's already sorted); otherwise split the list into two lists of roughly equal length, mergesort these two lists, then merge these two sorted lists.

Obviously we need helper functions for splitting and merging.

```
(* split : int list -> int list * int list    *)
(* REQUIRES true *)
(* ENSURES split(L) = a pair (A, B) of lists 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
```

Example: split $[1,2,3,4,5] = ([1,3,5],[2,4])$.

We can prove that split meets its specification, by induction on the length of the list being split. In the proof we appeal to some obvious facts about permutations.

- For L of length ≤ 1 the result holds obviously.
- Let L be a list of length $n > 1$. Then we can express L in the form x::y::R for some integers x and y and a list R of shorter length than n. By induction hypothesis, split R evaluates to a pair of lists (A, B) such that length(A) and length(B) differ by at most 1, and A@B is permutation of R. But then split L = (x :: A, y :: B), and these two lists have the same length difference as A and B; and (x::A)@(y::B) is a permutation of x::y::(A@B), hence also a permutation of L.
- That completes the proof.

Note that the spec is a bit more precise than the informal English that we used to introduce the algorithm.

The helper function for merging is only going to be used on a pair of sorted lists, and is used to produce another sorted list containing all of the items in both of the input lists. However, it would be terribly inefficient to call the `sorted` function here; we don't need to *check* that the inputs are sorted lists, provided we prove that the merge function only ever gets applied to pairs of sorted lists. And we can design the merge function so that it automatically builds a sorted result, so again there is no need to verify this property explicitly by calling `sorted`!

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

fun merge ([ ], B) = B
 |  merge (A, [ ]) = A
 |  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)
```

Examples:

```
   merge([1,3,5], [2,4]) = [1,2,3,4,5]
   merge([5,3,1], [2,4]) = [2,4,5,3,1]
```

We prove that `merge` meets its spec by induction on the sum of the lengths of `A` and `B`. This strategy should work, because in each recursive call the length of at least one of the two lists decreases by 1 (and the other one is either the same as before or shorter); in all cases the sum of the two list lengths is smaller. Here are the proof details. You might expect us to use as the "base case" in this proof the case where the sum of the list lengths is 0, i.e. when both lists are empty. However, because the of the way the function is written, it is actually simpler to base our inductive case analysis on the function definition, as follows.

Note that the spec asserts that `merge(A,B)` is *equal to* a sorted perm of `A@B`. This is the same as asserting that `merge(A,B)` *evaluates to* a sorted perm of `A@B`.

- Assume that `A` and `B` are sorted lists.

- If `A` or `B` is the empty list, then `merge(A,B)` returns `B` or `A`, respectively. In both cases the result is sorted, and since [ ]@`B` = `B` and `A`@[ ] = `A` in each case the result is equal to (and thus a permutation of) `A@B`.
- Inductive step: suppose that `A` and `B` are non-empty lists, and that `merge` satisfies the specification on all pairs of sorted lists whose length sum is smaller than that of `A` and `B`. Let `A` = `x` :: `A`$'$ and `B` = `y` :: `B`$'$. Just like the function definition (third clause), our proof branches on the result of comparing `x` and `y`.
  - If `x` $<$ `y`, `merge(A, B)` = `x` :: `merge(A`$'$`, B)`. By assumption that `x` :: `A`$'$ is sorted, `x` is $\leq$ every item in `A`$'$. And by assumption that `A` is sorted, so is `A`$'$. The length of `A`$'$ is one less than the length of `A`. So by the induction hypothesis, `merge(A`$'$`, B)` evaluates to a sorted list (say `M`) that is a permutation of `A`$'$`@B`. We have `merge(A, B)` = `x::M`. Since `x` $<$ `y` and we assumed that `y` :: `B`$'$ is sorted, `x` is $\leq$ every item in `B`. So `x` is $\leq$ every item in `M`. Hence `x` :: `M` is a sorted permutation of `x` :: (`A`$'$`@B`).
  - The case analysis for when `x` = `y` or `x` $>$ `y` is similar and we omit the details.
- We covered pairs in which one (or both) of the lists is empty in the first case; and the inductive step covers cases where both of the lists are non-empty. Thus we have shown by induction that for all sorted lists `A`,`B`, `merge(A, B)` evaluates to a sorted permutation of `A@B`.

Digression

The spec given here for `merge` only talks about what happens when the function gets applied to a pair of sorted lists. Of course the function can be used on any pair of integer lists whatsoever, but the specification doesn't tell us anything about what will happen. Here's what can happen:

```
val merge = fn : int list * int list -> int list
- merge ([2,1],[4,3]);
val it = [2,1,4,3] : int list
```

The input wasn't a pair of sorted lists and the output wasn't sorted. This doesn't clash with the spec, and doesn't cause any problems because we will only use the function by applying it to sorted lists!

Now we have the ingredients, we can define a mergesort function:

```
(* msort : int list -> int list *)
(* REQUIRES true *)
(* ENSURES  msort(L) = a sorted permutation of L *)

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

Note how closely the function definition resembles the informal algorithm description! And as promised, we've only used `merge` in a place where (it will be shown that) its arguments are sorted lists!

We now prove by induction on the length of `L` that `msort` meets this specification, for all integer lists `L`. Of course, we will make use here of the results (already established) that `split` and `merge` satisfy their specifications. Take note where (at the point labelled (*)) we establish that `merge` is being applied to sorted lists, which justifies our use of the proven spec for `merge`!

- Base case: When `L` is empty or a singleton list, `msort L` = `L`, and this is trivially a sorted list and a permutation of `L`.

- Inductive step: Assume that `L` is a list of length $n > 1$ and that `msort` satisfies the spec for lists of length less than n. From above, we know that `split L` evaluates to a pair of lists (`A`, `B`) such that $0 \leq$ `length`(`A`) $-$ `length`(`B`) ≤ 1 and `A@B` is a permutation of `L`. Hence, the length of `A` and length of `B` are both less than length of `L`. (The maximum possible length for `A` is n `div` 2 if n is even, n `div` $2 + 1$ if n is odd, and since $n > 1$ in each case this is less than n.) So, by the induction hypothesis, `msort A` evaluates to a sorted permutation of `A`, and `msort B` evaluates to a sorted permutation of `B`. So the values of `msort A` and `msort B` are sorted lists (*), and by the spec for `merge` it follows that `merge`(`msort A`, `msort B`) evaluates to a sorted permutation of `A@B`. This must also be a permutation of `L`, because "a permutation of a permutation is a permutation". End of proof.

Digression

Just for interest again, consider the following slight variant:

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

If we drop the singleton clause, like this, we get a function that loops on lists of length 1. Hence it also loops on any non-empty list. See where the above proof goes wrong if we try to use it to prove this code correct.

Efficiency of `msort`

The work (sequential running time) of `msort(L)` depends on the length of `L`, but not on the integers that occur in `L`. We can derive, from the function definition, a recurrence relation for the work $W_{\texttt{msort}}(n)$ of `msort(L)` when `L` has length n. To get an asymptotic estimate of the work for `msort`, we must also analyze the work of `split` and `merge`.

It seems pretty obvious that `split(L)` looks at each item in `L` successively, dealing them out into the left- or right-hand component of the pair of lists being constructed. So $W_{\texttt{split}}(n)$ is $\texttt{O}(n)$. We can reach the same conclusion by extracting a recurrence relation from the definition of `split`:

$$\begin{aligned}
&W_{\texttt{split}}(0) = 1\\
&W_{\texttt{split}}(1) = 1\\
&W_{\texttt{split}}(n) = 1 + W_{\texttt{split}}(n-2) \quad \text{for } n > 1
\end{aligned}$$

It is easy to show that the solution $W_{\texttt{split}}(n)$ to this recurrence relation is $\texttt{O}(n)$.

Similarly, when `A` and `B` are lists of length m and n, the running time of `merge(A, B)` is linear in $m + n$. (The output list has length $m + n$.)

Apart from the empty and singleton cases, `msort(L)` first calls `split(L)`, then calls `msort` recursively twice, each time on a list of length about half of the original list's length, then calls `merge` on a pair of lists whose lengths add up to `length(L)`. Hence, the work of `msort` on a list of length n is given inductively by:

$$\begin{array}{rcll} W_{\texttt{msort}}(0) & = & 1 & \\ W_{\texttt{msort}}(1) & = & 1 & \\ W_{\texttt{msort}}(n) & = & \texttt{W}_{\texttt{split}}(n) + 2W_{\texttt{msort}}(n \texttt{ div } 2) & \text{for } n > 1 \\ & = & \texttt{O}(n) + 2W_{\texttt{msort}}(n \texttt{ div } 2) & \end{array}$$

Using a table of standard solutions, or ootherwise, it follows that $W_{\texttt{msort}}(n)$ is $\texttt{O}(n\ log\, n)$. So the work for `msort` on a list of length n is $\texttt{O}(n\ log\, n)$.

What about the *span* for `msort L`? If we have the ability to run multiple processors to evaluate independent pieces of code, can we get a speed-up? You might think that the mergesort function is well suited for parallelism, because it makes two *independent* recursive calls on lists of half the length, used to build the two components of a pair. Hence calling `msort(L)` gives rise to a tree-shaped pattern of recursive calls, e.g.

```
           msort [3,4,1,2]
           /              \
   msort [3,1]           msort [4,2]
     /      \              /       \
msort [3]  msort [1]   msort [4]   msort [2]
```

and in general the height of this call tree is $O(\log n)$, where n is the length of the original list. So maybe mergesort has logarithmic span? Unfortunately, no! First we use `split` to deal the list out into two piles. There is no parallelism here, since we deal the elements out one by one, so we have to wait at least $\texttt{O}(n)$ timesteps in this phase, even if we have as much computational power as we need. This is bad. So the span of `split(L)` is linear in the length of `L`. For the same reason, the recurrence for the span of `split` is the same as the recurrence for the work of `split`, because the function is inherently sequential:

$$S_{\texttt{split}}(n) \quad = \quad 1 + S_{\texttt{split}}(n-2) \text{ for } n > 1$$

Thus, $S_{\texttt{split}}(n)$ is $\texttt{O}(n)$. Similarly, since `merge` is inherently sequential, the span of `merge` is (like the work) linear in the sum of the lengths of the lists. However, for `msort` we get, for $n \geq 2$,

$$\begin{array}{rcl} S_{\texttt{msort}}(n) & = & S_{\texttt{split}}(n) + max(S_{\texttt{msort}}(n \texttt{ div } 2), S_{\texttt{msort}}(n \texttt{ div } 2)) + S_{\texttt{merge}}(n) \\ & = & S_{\texttt{split}}(n) + S_{\texttt{msort}}(n \texttt{ div } 2) + S_{\texttt{merge}}(n) \\ & = & \texttt{O}(n) + S_{\texttt{msort}}(n \texttt{ div } 2) \end{array}$$

We use *max* here because the two recursive calls are independent, and can be calculated in parallel; since the recursive calls are both on lists of approximately half the length, we end up counting just one of them towards the span. We do need the additive terms for the span of split and the span of merge, because of the data dependencies: first the split happens, then the two parallel sorts, then the merge.

Expanding, and replacing the $\mathtt{O}(n)$ term by cn (for some positive constant c) to simplify our calculations, we see that:

$$\begin{aligned}
S_{\mathtt{msort}}(n) &= cn + S_{\mathtt{msort}}(n \;\mathtt{div}\; 2) \\
&= cn + cn/2 + cn/4 + cn/8 + cn/16 + \cdots + cn/2^{log_2 n} \\
&= cn(1 + 1/2 + 1/2^2 + \cdots + 1/2^{log_2 n}) \\
&\leq 2cn
\end{aligned}$$

The series sum here is always less than 2, and converges to 2 as n tends to infinity. (Technically, $\sum_{n=0}^{\infty} \frac{1}{2^n} = 2$.) So the span of `msort` is therefore $\mathtt{O}(n)$.

This is less than ideal. Ignore the constant factors, because a similar example can be chosen no matter what they are. Suppose you want to sort a billion numbers on 64 processors. Note that $\log 10^9$ is about 30, so the total work to do here is 30 billion steps. On 64 processors, this should take less than half a billion timesteps, if you divide the work perfectly among all 64 processors. However, our span estimate says that the length of the longest critical path is still a billion, so you can't actually achieve this division of labor! This problem gets worse as the number of processors gets larger.

The real issue here is that *lists are bad for parallelism.* The list data structure does not admit an efficient enough implementation of split and merge to exploit all the parallelism that might have been available.

In the next lecture we will discuss a more suitable data structure for parallel sorting.

6 The Joy of Specs

The mergesort example shows the benefits of designing helper functions with clear specifications, chosen carefully to make appropriate assumptions about the arguments to which the functions will be applied, and to make strong enough assertions about the results produced by these functions.

To illustrate the potential problems caused by inappropriate helper specs, note that the merge function also satisfies the specification:

```
For all integer lists L and R,
     merge(L,R) evaluates to a permutation of L@R.
```

This spec is not strong enough to help prove that `msort` sorts.

Note also that we can replace `split` by any other function with the same type that satisfies the same specification as we used above, without affecting the correctness of `msort` (defined as above, but using the replacement `split` function). The new split function doesn't need to be extensionally equivalent to the old one; it just needs to satisfy the same specification! For example we could have used

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

Exercises

- Check that this split function satisfies the same specification as before.
- Show that this function is not extensionally equivalent to the original `split`. (Hint: what is `split [1,2,3]`?) For which integer lists `L` do the two split functions produce exactly the same result?

7 Self-test

1. Let $T(n)$ for $n \geq 1$ be given by the following recurrence:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n \texttt{ div } 2) + n^2 \quad \text{for } n > 0 \end{aligned}$$

   Show that for all $k \geq 1$, $T(2^k) \leq 2^{2k+1}$. This is an easy induction! This result implies, by extrapolating, that $T(n)$ is $\texttt{O}(n^2)$.

   Contrast what happens here with the behavior of $U(n)$, given by

$$\begin{aligned} U(1) &= 1 \\ U(n) &= 2U(n \texttt{ div } 2) + n \quad \text{for } n > 0 \end{aligned}$$

   Show that for all $k \geq 1$, $U(2^k) \leq k2^{k+1}$.
   It follows that $U(n)$ is $\texttt{O}(n\ log\ n)$.

2. Prove by induction on the length of `L` that `sorted L = true` if every item in `L` is $\leq$ all later items in `L`, and `sorted L = false` otherwise. Indicate clearly when you appeal to the characteristic properties of $<$ as a linear ordering. Explain why the work to evaluate `sorted L` when `L` is a list value of length `n` is `O(n)`.

3. A self-classified smart friend(?) suggests that you implement a form of mergesort that checks first to see if the list is already sorted, to avoid all the recursive effort in that case. Here is their version:

```
fun silly_msort [ ] = [ ]
|   silly_msort [x] = [x]
|   silly_msort L = if (sorted L) then L else
  let
    val (A, B) = split L
  in
    merge (silly_msort A, silly_msort B)
 end
```

   What's the work for a list of length n? Is your friend smart, after all?

4. Another, rather insecure, friend suggests `new_msort`, which calls the old `msort` function only if it really needs to recurse (when the list is non-trivial and isn't already sorted).

   ```
   fun new_msort [ ] = [ ]
   |   new_msort [x] = [x]
   |   new_msort L = if (sorted L) then L else
     let val (A, B) = split L in merge (msort A, msort B) end
   ```

   What's the work for `new_msort L` when `L` is a list of length n? Any better, asymptotically, than your smart friend?

5. When the expression `ins(3,ins(2,ins(1,[ ])))` is evaluated, in what order do the insertions occur? How about `ins(1,ins(2,ins(3,[ ])))`?

6. What kind of argument values for `ins` cause the evaluation of `ins(x,L)` to take the most steps? These are called *worst-case* arguments for `ins`. Give a recurrence for the work of `ins(x,L)` when `L` is a list of length `n`. Solve the recurrence and give an asymptotic classification. Estimate the work for `isort L` when `L` is a list of length `n`.

7. The definition of `perm` given earlier has 4 clauses. Does it make any difference to the applicative behavior of the function if we change the clause order? Rewrite the `perm` function to use fewer than 4 clauses.

8. Write an ML function

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

   such that for all integer lists `L`, `split(L)` evaluates to a pair of lists $(\mathtt{A}, \mathtt{B})$ with $\mathtt{A@B} = \mathtt{L}$ and $|\mathtt{length(A)} - \mathtt{length(B)}| \leq 1$. Here $|x|$ means the absolute value of x. Would the code for `msort` still satisfy the sorting specification if we replace its `split` function with this one? How about the asymptotic work?