15-150 Lecture 6: Sequential Sorting

Lecture by Dan Licata

February 2, 2012

Today's lecture is about sorting. Along the way, we'll learn about *divide and conquer* algorithms, the *tree method*, and *complete induction*.

1 Mergesort

Sorting is a fundamental problem in computer science (e.g. when we want to enter your homework grades). It's simple to sort in quadratic time: see the notes on insertion sort below. Pick up one card, sort the rest of the pile, insert the card in the right place.

Mergesort is a better sorting algorithm, which has work $O(n \log n)$ instead of quadratic. This is much smaller for big n (and more-efficient enough that we use it to sort the homeworks).

Here's the idea:

```
          sort [3,1,4,2]
   sort [3,1]           sort [4,2]
sort [3]  sort [1]   sort [4]   sort [2]
```

At each step, divide the problem in half. Recursively solve these sub-problems. Then combine the results together.

Let's assume we've worked our way back up the tree to this point:

```
          sort [3,1,4,2]
     [1,3]               [2,4]
   [3]  [1]            [4]   [2]
```

How do we combine the results of sorting the recursive calls to get the overall result. The key operation is *merging* two sorted lists into a third one.

Because you can only divide a problem in half $\log n$ many times, there will be n levels in the tree. If we can manage to only do n work at each level, we'll have an $n \log n$ algorithm (we'll show this using the tree method later). So the question is:

So, the basic idea for mergesort is

- Split the list into two sublists of equal size
- Recursively mergesort them
- Merge the results back together

This is an instance of what is called a *divide and conquer* algorithm: you divide your problem into sub-problems, solve them recursively, and then combine the results back together. Usually the dividing and combining are written as helper-functions.

Here's a first cut at mergesort:

```
fun mergesort (l : int list) : int list =
   let val (pile1,pile2) = split l
   in
     merge (mergesort pile1, mergesort pile2)
   end
```

We spin the hard parts, splitting and merging, off into helper functions, and call them here (note that the helper functions would need to be above `mergesort` in the file). Now we have to implement the helper functions:

Splitting One way to split a list is to deal it into two piles of roughly equal size:

```
(* Purpose: deal the list into two piles p1 and p2,
            whose sizes differ by no more than 1.

            More formally: for all l, there exist (p1,p2) such that
            split l == (p1,p2) where p1@p2 is a permutation of l and
            the sizes of p1 and p2 differ by no more than one.
 *)
fun split (l : int list) : int list * int list =
    case l of
        [] => ([] , [])
      | [ x ] => ([ x ] , [])
      | x :: y :: xs => let val (pile1 , pile2) = split xs
                        in (x :: pile1 , y :: pile2)
                        end
val ([1,3],[2,4]) = split [1,2,3,4]
```

In the base cases, the sizes clearly differ by 0 and 1, respectively. In the cons-cons case, by the IH `pile1` and `pile2` differ by no more than one, and the results add 1 to each.

Merging

```
(* Purpose: merge two sorted lists into one sorted list,
            containing all and only the elements of both *)
fun merge (l1 : int list , l2 : int list) : int list =
    case (l1 , l2) of
        ([] , l2) => l2
      | (l1 , []) => l1
      | (x :: xs , y :: ys) =>
            (case x < y of
                 true => x :: (merge (xs , l2))
```

```
              | false => y :: (merge (l1 , ys)))
val [1,2,3,4] = merge ([1,3],[2,4])
```

If either list is empty, the merge is the other one. Otherwise, the smaller of the two first elements is the smallest element of both: e.g. suppose `x < y` is true. Then `x` is less than everything in `xs` because `x::xs` is sorted, and `y` is less than everything in `ys` because `y::ys` is sorted, and so by transitivity `x` is less than everything in `ys`.

Merge illustrates more general *pattern-matching* than we've seen so far. The general form of a `case` is

```
case e of
   p1 => e1
 | p2 => e2
 ...
```

where each `p` is a pattern. What are patterns?

pattern	type	matches
x	any	anything, binding it
_	any	anything
1,2, 1,...	`int`	that numeral
`[]`	`int list`	`[]`
`p1::p2`	`int list`	a list `v1::v2` where `v1` matches `p1` and `v2` matches `p2`
`(p1,p2)`	`A * B`	a pair `(v1,v2)` where `v1` matches `p1` and `v2` matches `p2`

Note that you can't use an arbitrary expression as a pattern. The reason is to enable ML to do *exhaustiveness* and *redundancy* checking. If we forgot a case:

```
case (l1 , l2) of
    ([] , l2) => l2
  | (x :: xs , y :: ys) =>
```

ML would say that this pattern-match is non-exhaustive (doesn't cover all values. If we had an extra case:

```
case (l1 , l2) of
    ([] , l2) => l2
  | (l1, []) => l1
  | ([], []) => l1
  | (x :: xs , y :: ys) =>
```

ML would say that the extra case is redundant (can never be reached). These checks are extremely useful for software evolution, because when you change a type, you get warnings about all of the functions you need to change.

Mergesort So we're done with mergesort, right?

In pairs, try it on `mergesort [2,1]`. When you have correctly stepped this expression to a value, you are free to leave for the day =).

Why are you all still here?

```
     mergesort [2,1]
|->* merge (mergesort [2] , mergesort [1])
|->* merge (merge (mergesort[2] , mergesort []) , mergesort [1])
|->* merge (merge (merge (mergesort[2] , mergesort []) , mergesort []) , mergesort [1])
|->* ...
```

Ooops! It's an infinite loop!

What went wrong? The problem is that the lists returned by `split` are not necessarily smaller than the input! `split [1] ==> ([1],[])`.

However, they are smaller as long as the list has two or more elements! If the sizes are the same, then each pile has $n/2$ elements, and if $n \geq 2$, then $n/2$ is less than n. If the sizes differ by 1, then one pile has $n/2$ elements and the other $n/2 + 1$, which are both smaller for $n \geq 3$.

So we just need to special-case the empty and singleton lists:

```
fun mergesort (l : int list) : int list =
    case l of
        [] => []
      | [x] => [x]
      | _ => let val (pile1,pile2) = split l
             in
                 merge (mergesort pile1, mergesort pile2)
             end
val [1,2,3,4,5] = mergesort [5,3,2,1,4]
```

Mergesort is our first example of a function that is *not structurally recursive*: it doesn't follow the template for recursion on lists. The moral: if you compute recursive call inputs in an interesting way, you have to carefully check that they are in fact smaller! Otherwise, you might write a function that's not total, like the original version of `mergesort`.

1.1 Work

If n is the length of the input lists, then the work for mergesort is $O(n \cdot \log n)$. This is much better than the $O(n^2)$ insertion sort: if you need to sort a billion things, then it takes roughly 30 Xbillion steps, not 10^18.

First, we extract a recurrence relation for `split`, letting n stand for the size of the input.

$$\begin{aligned} W_{split}(0) &= k_0 \\ W_{split}(1) &= k_0 \\ W_{split}(n) &= k_2 + W_{split}(n-2) \end{aligned}$$

In each base case, we do some constant amount of work. In the remaining case, we do some constant amount for work around the recursive call. This recurrence is $O(n)$. To see this, we can observe that the closed form is

$$\sum_{i=1}^{n/2}(k_2) + k_0 = k_3 * n + k_0$$

Counting down by 2's is linear, just like counting down by 1's.

For merge, if we let n be the length of l_1 and m be the length of l_2, then

$$\begin{aligned} W_{merge}(0,m) &= k_0 \\ W_{merge}(n,0) &= k_0 \\ W_{merge}(m,n) &= k_1 + (\text{ either } W_{merge}(m-1,n) \text{ or } W_{merge}(m,n-1)) \end{aligned}$$

The either/or is hard to deal with, so we can change the number we are analyzing in terms of to the sum $s = m + n$. Then we have:

$$\begin{aligned} W_{merge}(0) &= k_0 \\ W_{merge}(s) &= \text{ either } k_0 \text{ or } (k_2 + W_{merge}(s-1)) \end{aligned}$$

The k_0 comes when one of the two lists is empty; the $(k_2 + W_{merge}(s-1))$ when they're both conses.

Usually we will do *worst-case analysis*: this means we analyze the running time of the function, assuming the inputs are as bad as possible. So in a situation like this, we assume the worst:

$$\begin{aligned} W_{merge}(0) &= k_0 \\ W_{merge}(s) &= k_2 + W_{merge}(s-1) \end{aligned}$$

Now it's clear that W_merge is $O(n+m)$.

For mergesort, we have therefore have

$$\begin{aligned} W_{mergesort}(0) &= k_0 \\ W_{mergesort}(1) &= k_0 \\ W_{mergesort}(n) &= k_1 + W_{merge}(n/2 + n/2) + W_{split}(n) + 2 \cdot W_{mergesort}(n/2) \\ &\leq k_2 n + 2 \cdot W_{mergesort}(n/2) \end{aligned}$$

We make the slight simplifying assumption that the list has even length; otherwise the math is a little messier but the big-O is the same. Each recursive call is on a list of length $n/2$ because split returns two lists of equal size. Further, each argument to `merge` has length $n/2$ because the output of mergesort has the same length as the input, which is the output of split. Because W_{merge} and W_{split} are $O(n)$, we can plug in and coalesce:

$$W_{mergesort}(n) \leq kn + 2 \cdot W_{mergesort}(n/2)$$

Each call to `mergesort` does linear work in addition to the recursive calls. We can obtain a solution for a recurrence of this form using the *tree method*:

1. write out the recursive calls as a tree
2. sum each level of the tree
3. sum the levels

In the first step, we get

```
          kn
   kn/2          kn/2
kn/4 kn/4     kn/4 kn/4
...
```

Because we're summing this all up, we can factor k out, and multiply it times the sum of

```
          n
   n/2         n/2
n/4 n/4     n/4 n/4
...
```

To sum this tree, we can sum the levels:

```
          n           n
   n/2         n/2     n
n/4 n/4     n/4 n/4    n
...
```

In this case, each level sums to n! Thus, the sum of all of the levels is just n times the number of levels. How many levels are there? The depth of the tree is the number of times you can divide n in half before getting to 1, which is $\log n$. So there are $\log n$ rows, each of which requires n work, which means the whole thing is $(n \log n)$. Thus, the closed form is

$$k(n \log n) + k_0$$

which is $O(n \log n)$.

1.2 Correctness

Please read through the following material, so that you can do a proof that needs complete induction if we ask you to.

Let's get more formal about why `mergesort` is correct. Along the way, we'll meet some new induction principles.

First, we need to define what it means for a list to be sorted. esorted is a predicate on expressions of type `int list` defined as follows:

- `[]` sorted
- `e::es` sorted **iff** `e` valuable and `es` sorted and $(\forall x \in e2, e1 \leq x)$
- `e` sorted if `e` $\cong$ `e'` and `e'` sorted

The empty list is sorted. A cons is sorted if the tail is sorted and the head is less than or equal to everything in it. Moreover, sortedness respects equivalance. The condition on a cons is also an **if and only if**: if we know that a cons is sorted, then we know that the tail is sorted, etc.

You might wonder how we define such a predicate: technically, what you do is define it with the cons condition as an "if" and then prove the "only if" as a lemma (which is a little tricky because of the third clause about equivalence). But to make things easy we've hidden this behind the scenes.

It's simple to see that sorted expressions are valuable:

Lemma 1. *If* `e` sorted *then* `e` valuable.

Correctness of `mergesort` Let's prove that `mergesort` returns a sorted list:

Theorem 1. *For all values `l:int list`, `(mergesort l)` sorted.*

Clearly we're going to need some lemmas about the helper functions, but let's do the proof outside-in and see what we need. The important idea here is that *the structure of the proof should follow the structure of the code:*

Proof. **Case for `[]`:**
To show: `(mergesort [])` sorted.
Proof: `mergesort []` $\cong$ `[]` (by 2 steps), `[]` is sorted, and sorted respect equivalence.

Case for `[x]`:
To show: `(mergesort [x])` sorted.
Proof: `mergesort [x]` $\cong$ `[x]` (by 2 steps). `[x]` is sorted because

1. `x` valuable: variables stand for values, and values are valuable
2. `[]` sorted
3. $\forall y \in [], x \leq y$. This is *vacuously true* because there are no elements of the empty list.

and sorted respects equivalence.

Case for other `l`:
To show: `(mergesort l)` sorted.
Proof:

```
   mergesort l
== let val (p1,p2) = split l in merge (mergesort p1, mergesort p2) end  (2 steps)
```

Here's how we'd like to reason:

1. `split l` returns two smaller lists `p1` and `p2`.
2. Inductive hypotheses tell us that the recursive calls are sorted.
3. `merge` produces a sorted list when give two sorted lists.

So let's invent lemmas that give us what we want (see below). Then we can say:

1. By Lemma 2, there exist `p1,p2` such that `split l` $\cong$ `(p1,p2)` where `p1` $\sqsubset$ `l` and `p2` $\sqsubset$ `l` (note that `l` is not `[]` or `[x]` in this case).
2. To make the induction work, we do *complete induction on lists*. This means the inductive hypothesis is that all smaller lists are sorted:

   $$\forall \texttt{l'} \sqsubset \texttt{l}, \texttt{(mergesort l')}\ \text{sorted}$$

   This is necessary because `mergesort` recurs on arbitrary smaller lists: complete recursion in the code corresponds to complete induction in the proof.

   Because `p1` $\sqsubset$ `l` and `p2` $\sqsubset$ `l`, the IH tells us in particular that `(mergesort p1)` sorted and `(mergesort p2)` sorted.
3. By Lemma 5 below, `(merge(mergesort p1, mergesort p2))` sorted. Note that `mergesort p1` and `mergesort p2` are valuable because they are sorted, by Lemma 1.

□

Lemma about split Define `l ⊏ l'` to mean `l` is obtained from `l'` by deleting one or more elements. E.g. `[1,3,5] ⊏ [1,2,3,4,5]`.

Lemma 2. *For all values `l:int list`, there exist values `p1,p2:int list` such that `split l` ≅ `(p1,p2)`, and if `l` is not `[]` or `[x]` (for some x), then `p1` ⊏ `l` and `p2` ⊏ `l`.*

The proof is simple, but needs complete induction on `l`, because `split` recurs on `xs` in the case for `x::y::xs`.

Lemma about merge

Lemma 3. *For all values `l1 l2:int list`, if `l1`*sorted *and `l2`*sorted *then `(merge(l1,l2))`*sorted*.*

Proof. **Case for `([],l2)`:**
To show: `(merge ([],l2)` sorted.
Proof: `merge ([],l2)` ≅ `l2` which is sorted by assumption.
Case for `(l1,[])`:
To show: `(merge (l1,[])` sorted.
Proof: `merge (l1,[])` ≅ `l1` which is sorted by assumption.
Case for `(x::xs,y::ys)`:
To show: `(merge (x::xs,y::ys)` sorted.
Proof:

```
   merge (x::xs,y::ys)
== (case x < y of true => x :: (merge (xs , l2))
                | false => y :: (merge (l1 , ys)))
```

When you have a `case` in your code, you need a case in your proof: `x < y` is valuable, so we have two cases:

- `x < y` ≅ `true`. Then

```
   merge (x::xs,y::ys)
== (case x < y of true => x :: (merge (xs , y::ys))
                | false => y :: (merge (x::xs , ys)))   [2 steps]
== (case true  of true => x :: (merge (xs , y::ys))
                | false => y :: (merge (x::xs , ys)))   [assumption]
== x :: (merge (xs , y::ys))                            [step]
```

  So it suffices to show that `x::merge(xs,l2)` is sorted. We need to show three things:

  1. `x` valuable: variables stand for values.
  2. `(merge(xs,l2))` sorted. This should be true by the IH. But what justifies this inductive call? The key idea is something called *simultaneous induction*: a pair is smaller if one component gets smaller and the other stays the same. In this case, `xs` ⊏ `x::xs` so `(xs,y::ys)` ⊏ `(x::xs,y::ys)`. So we can use the inductive hypothesis on `(xs,y::ys)`. `y::ys` is sorted by assumption. `xs` is sorted because `x::xs` is (this is where we use the **iff** in the definition of sorted). Thus, by the IH, `merge(xs,y::ys)` is sorted.

3. $\forall$ `z` $\in$ `merge(xs,y::ys)`, `z < x`. Because `x::xs` is sorted, `x` is less than every element of `xs`. Because `x < y` $\cong$ `true`, `x < y`. Because `y::ys` is sorted, `y` is less than every element of `ys`, so by transitivity, `x` is less than every element of `ys`. Thus, we know that `x` is less than every element of `xs@(y::ys)`. Thereforem, Lemma 4, which states that `merge` returns a permutation of its input, gives the result, because the elements of a permutation of a list are the same as the elements of the list.

- `x < y` $\cong$ `false`. Analogous to the above; left as an exercise.

□

Lemma 4. *For all values `l1 l2: int list`, `merge(l1,l2)` is a permutation of `l1@l2`.*

Proof. Simultaneous induction. Details left as an exercise. □

We usually state theorem about values because it makes them easy to prove by induction: the induction principle for lists applies to *values*, not *expressions*. But we can lift lemmas to valuable expressions:

Lemma 5. *For all* ***valuables*** *`e1 e2:int list`, if* `e1`sorted *and* `e2`sorted *then* `(merge(e1,e2))`sorted.

Proof. In general, if you know "for all values `x`, P(`x`)" and P respects equivalence, then P holds for all valuable expressions `e`: Because `e` is valuable, `e` $\cong$ `v`. Because `v` is a value, P(`v`). Because P respects equivalence, P(`e`). □

2 Reading: Insertion Sort

Note, Spring 2012: we didn't go over insertion sort in lecture, since it's not that much harder than the reverse code from Lecture 5.

Let's sort a list so that all the numbers are increasing order. That is we want to write a function `isort` such that

```
isort : int list -> int list
isort [1,3,2,4] ==> [1,2,3,4]
```

Here's a strategy for writing recursive functions on lists according to the pattern of *structural recursion*:

```
fun isort (l : int list) : int list =
  case l of
     [] => ...
   | x :: xs => ... sort xs ...
```

First, case-analyze the list, distinguish cases for empty and `x::xs`. In the latter case, you will likely make a recursive call on `xs`.

Now, to fill in the template, what you want to do is *assume the function works for the recursive call, and then figure out what's left to do.* Soon, you'll be proficient enough at doing this to think very abstractly, in terms of the spec of the function. While you're learning, it's best to do this in terms of an example.

E.g., we want

```
isort [3,1,4,2] ==> [1,2,3,4]
```

In the case for this input, which can also be written as `3 :: [1,4,2]`, we're going to make a recursive call on `[1,4,2]`. So assume this recursive call does the right thing:

```
isort [1,4,2] ==> [1,2,4]
```

And now, what's *left to do*, to get from the result of the recursive call, to the result on `[3,1,4,2]`? We need to put `3` in the right place.

It's often helpful to break this bit, that takes you from the result of the recursive call, to the overall result, out into a helper function. Here, we can write:

```
(* Example: insert(3,[1,2,4]) ==> [1,2,3,4] *)
fun insert (n : int , l : int list) : int list = ...

fun isort (l : int list) : int list =
    case l of
        [] => []
      | (x :: xs) => insert (x , isort xs)
```

The sorted version of the empty list is the empty list. The sorted version of `x :: xs` is the sorted version of `xs`, with `x` inserted in the appropriate place.

Do `insert` in pairs. Here's the final result:

```
(* Purpose: insert n in sorted order into l.
            If l is sorted, then insert(n,l) ==> l' where
               l' is sorted
               l' is a permutation of n::l
*)
fun insert (n : int , l : int list) : int list =
    case l of
        [] => n :: []
      | (x :: xs) => (case n < x of
                          true => n :: (x :: xs)
                        | false => x :: (insert (n , xs)))

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

To `insert` the first element, we walk down the the list. If we get to the end, then insert it there. Otherwise, check whether it should be the first element. If so, insert it, because it's less than the head, which is in turn less than the rest of the list, by the assumption that it's sorted. If not, recursively insert it in the right place in the rest, and don't forget to put the head back on.

Here's the final version of `isort`:

```
(* Purpose: sort l
            For all l, isort l ==> l' where
              l' is sorted (in increasing order) and a permutation of l
*)
```

```
fun isort (l : int list) : int list =
    case l of
        [] => []
      | (x :: xs) => insert (x , isort xs)

val [ 1 , 2 , 3 , 4 , 5] = isort [ 5, 1 , 4 , 3, 2]
```

Analysis What's the work for insert?

$$\begin{array}{ll} W_{\texttt{insert}}(0) = & k_0 \\ W_{\texttt{insert}}(1) = & k_1 \\ W_{\texttt{insert}}(n) = & k_2 + W_{\texttt{insert}}(n-1) \text{when we have to insert into the rest} \\ & k_3 \text{when we find the right place} \end{array}$$

This is the first opportunity we have to talk about *worst-case analysis*. This means we analyze the running time of the function, assuming the inputs are as bad as possible. So in a situation like this, we assume the worst:

$$\begin{array}{ll} W_{\texttt{insert}}(0) = & k_0 \\ W_{\texttt{insert}}(1) = & k_1 \\ W_{\texttt{insert}}(n) = & k_2 + W_{\texttt{insert}}(n-1) \end{array}$$

As you know, a recurrence of this form is $O(n)$.

For insertion sort we have:

$$\begin{array}{ll} W_{\texttt{isort}}(0) = & k_0 \\ W_{\texttt{isort}}(n) = & k_1 + W_{\texttt{insert}}(n-1) + W_{\texttt{isort}}(n-1) \end{array}$$

Because `insert` is $O(n)$, this means

$$W_{\texttt{isort}}(n) \leq k_1 + n * k_2 + W_{\texttt{isort}}(n-1)$$

and a recurrence of this form is $O(n^2)$: you do n work n times.