Lesson 7
SORTING AND PARALLELISM

June 6, 2023

Last time, we reviewed the idea of **asymptotic analysis**, which is the analysis of the performance of programs, as the input size grows.

We learned that for recursive, functional programs, we could write mathematical **recurrences** that described the `work` of the code, and that could be solved via the **unrolling** method to obtain a closed form, and a bound.

We also learned that, by assuming we had infinitely many processors, we could obtain recurrences that measured the `span` of the code, or the amount of time using parallelism. We saw that this gave performance benefits for `treesum` on balanced trees.

1 - Analyzing a Tree via Depth

Before, we discussed how we could use the number of nodes in a tree as the input size, and obtain two span recurrences for `treesum` – $O(n)$ in the imbalanced case, and $O(\log n)$ in the balanced case.

That's not the only way to measure a tree, though.[1] The other way is that we could use the **depth** of the tree, which is the longest path through the tree to get to the bottom.

```
fun treesum (Empty : tree) : int = 0
  | treesum (Node (L, x, R)) = treesum L + x + treesum R
```

Let's try it!

[1] You can also use a tape measure.

Where d is the depth of the tree `T`, in the expression `treesum T`:

$$S_{\texttt{treesum}}(0) = c_0$$

$$S_{\texttt{treesum}}(d) = \max(S_{\texttt{treesum}}(d_{\texttt{L}}), c_1, S_{\texttt{treesum}}(d_{\texttt{R}})) + c_2$$ [2]

Again, we don't know how deep the tree is in the left and right subtrees – our recurrence depends on the quantities $d_{\texttt{L}}$ and $d_{\texttt{R}}$, respectively.

We find ourselves in the same situation as before, and we'll solve it the same way, by assuming the worst case. In the worst case, the tree is just a spine, so the depth of the left is $d - 1$, and the depth of the right is 0.

[2] Recall that the c_1 term is obtained via the constant amount of work involved in the computation of `x`. It's already a value, but there is *some* constant work associated, and we do take the max over it, in case it unexpectedly takes a really long time.[3]

[3] It doesn't.

Case The **span** of `treesum` on an **unbalanced** tree, in terms of the **depth** d.

$$S_{\texttt{treesum}}(d) = \max(S_{\texttt{treesum}}(d-1), c_1, S_{\texttt{treesum}}(0)) + c_2$$

$$S_{\texttt{treesum}}(d) = S_{\texttt{treesum}}(d-1) + c_2$$

If we exchange d for n, we've seen this recurrence before. This solves to $O(d)$.

$$\begin{aligned}
&S_{\texttt{treesum}}(d) \\
&= S_{\texttt{treesum}}(d-1) + c_2 \\
&= S_{\texttt{treesum}}(d-2) + c_2 + c_2 \\
&= \ldots \\
&= \sum_{i=1}^{d} c_2 + c_0 \\
&= d \cdot c_2 + c_0
\end{aligned}$$

Case The **span** of `treesum` on a **balanced** tree, in terms of the **depth** d.

Then, the left subtree still has depth $d-1$, as does the right subtree.

So:

$$S_{\texttt{treesum}}(d) = \max(S_{\texttt{treesum}}(d-1), c_1, S_{\texttt{treesum}}(d-1)) + c_2$$

$$S_{\texttt{treesum}}(d) = S_{\texttt{treesum}}(d-1) + c_2$$

What gives?? This is the same recurrence! Span is $O(d)$ in both the unbalanced and balanced cases.

This might be counterintuitive, because we expect a better bound, but it makes sense if you remember the task dependency graphs we discussed earlier.

In a task dependency graph, the cost of executing some amount of tasks in parallel is just the *longest path through the graph*, or tree. The length of the longest path through a tree is just d, the depth of the tree, because a tree has the structure of a dependency graph that is just the same tree!

We can relate it back to our previous bounds, $O(n)$ and $O(\log n)$, for unbalanced and balanced trees, in the number of nodes n, however.

Key In an unbalanced tree, the depth d is just the number of nodes n.

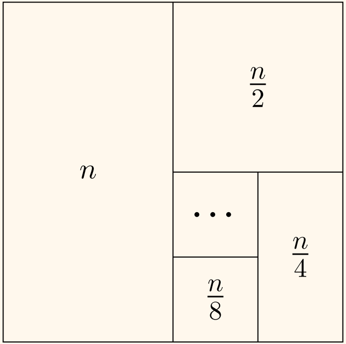
So in the unbalanced case, an $O(d)$ bound is the same as $O(n)$, which is the same bound we received earlier.

What is the number of nodes in a balanced tree of depth d though? Well, each level has double the nodes of the previous, so it's equal to

$$1 + 2 + 4 + ... + 2^d$$

Key There's a lovely geometric proof that shows that this is in $O(2^d)$.

So in a balanced tree, $n \approx 2^d$, so our previous bound is $O(\log n) = O(\log(2^d)) = O(d)$. We get the same thing either way, so this is perfectly consistent!



We can see that if we do an infinite sum of $n + \frac{n}{2} + \frac{n}{4} + \ldots$, the sum never surpasses the area of the square, which is just twice of the left half, otherwise known as $2n$.

Since the finite sum $1 + 2 + 4 + \ldots + n$ is definitely smaller than this, we are fine to conclude that it is on the order of $O(n)$.

Ultimately, if you do the math and reason it out, you find that getting bounds in terms of depth and nodes looks different, but ultimately say the same thing.

Whichever is "easier" is up to your discretion. Both are valid ways of solving a recurrence.[4]

Span of `treesum`	Nodes	Depth
Balanced	$O(\log n)$	$O(d)$
Unbalanced	$O(n)$	$O(d)$

[4] We will usually specify whenever we have a particular way we want to see you solve it, which is often.

2 - The Tree Method

Recall our notion of an *traversal* on a tree, which produces a list from a tree by traversing the tree in some prescribed order.

We are interested in *inorder* traversal, which traverses a tree the same way that someone would traverse it by reading from left-to-right.

```
fun inord (Empty : tree) : int list = []
  | inord (Node (L, x, R)) = inord L @ (x :: inord R)
```

When you see recursive calls being given as arguments to append, you should double-check, because something fishy is probably going on.

But, better than thinking about it, we can mathematically solve for the performance! Let's assume a balanced tree, and solve for the work of this function, in terms of the nodes of the tree.

Case The **work** of `inord` on a **balanced** tree.

Where n is the number of nodes in the input tree:

$$W_{\texttt{inord}}(0) = c_0$$

$$W_{\texttt{inord}}(n) = c_1 + W_{\texttt{@}}\left(\frac{n}{2}\right) + 2 \cdot W_{\texttt{inord}}\left(\frac{n}{2}\right)$$

(because we append a list of half the size, and compute `inord` recursively twice)

So we have:

$$
\begin{aligned}
& W_{\texttt{inord}}(n) \\
& = c_1 + W_{\texttt{@}}\left(\frac{n}{2}\right) + 2 \cdot W_{\texttt{inord}}\left(\frac{n}{2}\right) \\
& = c_1 + O(n) + 2 \cdot W_{\texttt{inord}}\left(\frac{n}{2}\right) \\
& = c_1 + O(n) + 2 \cdot \left(c_1 + O\left(\frac{n}{4}\right) + 2 \cdot W_{\texttt{inord}}\left(\frac{n}{4}\right)\right) \\
& = c_1 + O(n) + 2 \cdot \left(c_1 + O\left(\frac{n}{4}\right) + 2 \cdot \left(c_1 + O\left(\frac{n}{8}\right) + 2 \cdot W_{\texttt{inord}}\left(\frac{n}{8}\right)\right)\right) \\
& = \ldots = ???
\end{aligned}
$$

This is... messy.[5]

[5] Life is, sometimes.

Sometimes, unrolling is messy. Sometimes, like in `length`, we only get one extra term per "unrolling", and so it's not hard to solve by just finding the pattern.

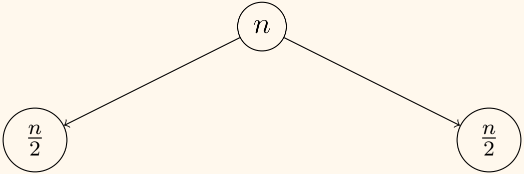
In the case of functions on trees, this usually isn't the case! We actually get *two* terms per unrolling, which quickly becomes four by the next unrolling, and so on. This is much harder to eyeball.

We will employ a new technique of solving for such recurrences, using the **tree method**, instead of the unrolling method.

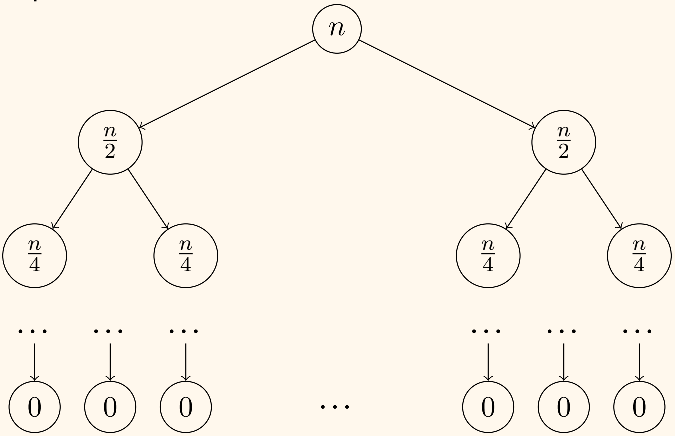
The tree method gets its name, from noticing that the amount of recursive calls done by a function like `inord` induces a tree structure.

We see that calling `inord` on a tree with n nodes causes two calls, to `inord` with $\frac{n}{2}$ nodes (in the balanced case).

So let's draw a tree representing the computation of `inord T`, with nodes annotated with the size of the input at each call, with edges indicating recursive calls:



But those two calls to `inord` have their own recursive calls, which have size $\frac{n}{4}$.
So our "call tree" expands to:



We can think of the following recurrence as three parts:

$$W_{\texttt{inord}}(n) = c_1 + W_{\texttt{@}}(\frac{n}{2}) + 2 \cdot W_{\texttt{inord}}(\frac{n}{2})$$

- the *nonrecursive work*, $W_{\texttt{@}}(\frac{n}{2}) + c_1$
- the *recursive work*, $2 \cdot W_{\texttt{inord}}(\frac{n}{2})$
- for a given *input size*, which is n

With respect to the call tree, the nonrecursive work is the work present at each node, but the recursive work is taken care of by all its children.

So, if we sum all the nonrecursive work in **each node**, we'll get the work done by the entire function.

$$W_{\texttt{inord}}(n) = c_1 + W_{\texttt{@}}(\frac{n}{2}) + 2 \cdot W_{\texttt{inord}}(\frac{n}{2})$$

To do this, we'll need to somehow figure out the nonrecursive work done by each node. This is $O(n) + c_1$ for a node with size n, except each node has a different size!

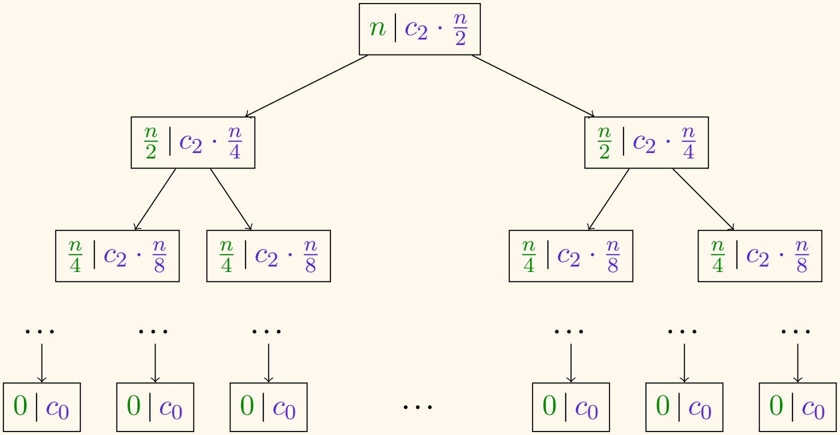
In addition, there's a differing number of nodes of each size, since there's 2 of size $\frac{n}{2}$, and 4 of size $\frac{n}{4}$, and so on.

This isn't easier at all!

The innovation comes from noticing that the nonrecursive work at each *level* of the tree might come out to the same thing.

If the work at each level was the same, then we could just multiply that quantity by the height of the tree, which is $\log n$, the number of times we can recursively call `inord` by halving the input.

Let's try it out.



$$= 2^0 \cdot c_2 \cdot \frac{n}{2} = c_2 \cdot \frac{n}{2}$$

$$= 2^1 \cdot c_2 \cdot \frac{n}{4} = c_2 \cdot \frac{n}{2}$$

$$= 2^2 \cdot c_2 \cdot \frac{n}{8} = c_2 \cdot \frac{n}{2}$$

$$= \frac{n}{2} \cdot c_0 = c_0 \cdot \frac{n}{2}$$

where the green denotes the size of the input at a node, and the purple denotes the amount of nonrecursive work at that call

Let the *level* of a tree denote how far we are from the root. So the root is at level 0, and there are two nodes at level 1, and so on.

Number of levels $\log n$, the number of times we can divide the input size by 2

Work per node at level i $\frac{n}{2^{i+1}} c_2$

Number of nodes at level i 2^i

So to solve for our cumulative work at level i, we multiply the number of nodes and work per node:

$$2^i \cdot \left(\frac{n}{2^{i+1}} c_2\right) = \frac{n}{2} \cdot c_2$$

So the work at level i, cumulatively, is the same! What's more, it's in $O(n)$. There's $\log n$ levels, so we ultimately come out with a bound of $O(n \log n)$. [6]

[6] There are several things we elided to come to this bound. We decided to count the nonrecursive work at each node as the work of asymptotically dominating `@` and we ignored the c_0 leaf terms, because they are ultimately dominated by the sum of the other layers. We only care about getting to the right asymptotic bound, so we can make things disappear if they aren't relevant.

Complexity of `inord`	Work	Span
Balanced	$O(n \log n)$	TBD
Unbalanced	TBD	TBD

So now we've filled in one entry for our matrix of work and span for the `inord` function, in the balanced and unbalanced case.

Let's quickly reason about the unbalanced case for the work.

Case The **work** of `inord` on an **unbalanced** tree.

The worst case would be if the subtree `L` had $n-1$ nodes, in other words a left spine. This is because we would have to do an append of $n-1$ elements.

Where n is the number of nodes in the input tree:

$$W_{\texttt{inord}}(0) = c_0$$

$$W_{\texttt{inord}}(n) = c_1 + W_{\texttt{@}}(n-1) + W_{\texttt{inord}}(n-1) + W_{\texttt{inord}}(0)$$

This looks very similar to our `rev` recurrence from last lecture. We will skip the derivation and conclude that the complexity is $O(n^2)$.

Case The **span** of `inord` on a **balanced** tree.

```
fun inord (Empty : tree) : int list = []
  | inord (Node (L, x, R)) = inord L @ (x :: inord R)
```

Here, we can do the calls to `inord L` and `inord R` in parallel, so we get the same recurrence as the balanced work case, but with only one recursive call.

Where n is the number of nodes in the input tree:

$$S_{\texttt{inord}}(0) = c_0$$

$$S_{\texttt{inord}}(n) = \max\left(S_{\texttt{inord}}\left(\frac{n}{2}\right), c_1, S_{\texttt{inord}}\left(\frac{n}{2}\right)\right) + S_{\texttt{@}}\left(\frac{n}{2}\right) + c_2$$

$$
\begin{aligned}
& S_{\texttt{inord}}(n) \\
& = \max\left(S_{\texttt{inord}}\left(\frac{n}{2}\right), c_1, S_{\texttt{inord}}\left(\frac{n}{2}\right)\right) + S_{\texttt{@}}\left(\frac{n}{2}\right) + c_2 \\
& = S_{\texttt{inord}}\left(\frac{n}{2}\right) + \frac{n}{2}c_1 + c_2 \\
& = S_{\texttt{inord}}\left(\frac{n}{4}\right) + \frac{n}{4}c_1 + c_2 + \frac{n}{2}c_1 + c_2 \\
& = \ldots \\
& = \sum_{i=1}^{\log n}\left(\frac{n}{2^i}c_1\right) + \log n \cdot c_2 + \frac{n}{2}c_0
\end{aligned}
$$

The first term dominates, because it's $(1 + 2 + 4 + 8 + \ldots + \frac{n}{2})c_1$, so we get $O(n)$.

Complexity of `inord`	Work	Span
Balanced	$O(n \log n)$	$O(n)$
Unbalanced	$O(n^2)$	$O(n^2)$

We leave it as an exercise to the reader that the span of the unbalanced `inord` case is $O(n^2)$.

Recall that appending recursive calls typically denotes something fishy. Let's try to think and see if we can eliminate that, in favor of a better work complexity than $O(n \log n)$.

3 - A Better `inord`

Let's do `inord` again, but this time with an accumulator argument. Let's try to avoid using `@` with a recursive call.

```
fun inord' (Empty : tree, acc : int list) = acc
  | inord' (Node (L, x, R), acc) =
      inord' (L, x :: inord' (R, acc))
```

Theoretically, the complexity should be better. Let's figure it out!

Case The **work** of `inord'` on a **balanced** tree.

Where n is the number of nodes in `T` in the expression `inord' (T, L)`:

$$W_{\texttt{inord'}}(0) = c_0$$

$$W_{\texttt{inord'}}(n) = 2 \cdot W_{\texttt{inord'}}\left(\frac{n}{2}\right) + c_1$$

We get two calls to $W_{\texttt{inord'}}\left(\frac{n}{2}\right)$, because we first compute `inord'(R, acc)`, and then pass that in as `acc'` to `inord' (L, x :: acc')`.

In either case, the size of the tree being called on is roughly half.

So we solve to:

$$
\begin{aligned}
&= W_{\texttt{inord'}}(n) \\
&= 2 \cdot W_{\texttt{inord'}}(\frac{n}{2}) + c_1 \\
&= 4 \cdot (W_{\texttt{inord'}}(\frac{n}{4}) + c_1) + c_1 \\
&= \ldots
\end{aligned}
$$

Same issue as before, now we have two recursive calls at each unrolling. Better to solve this with the tree method!

$$W_{\texttt{inord'}}(n) = 2 \cdot W_{\texttt{inord'}}(\frac{n}{2}) + c_1$$

Number of levels $\log n$

Work per node at level i c_1

Number of nodes at level i 2^i

So then our cumulative work at level i is just the product of the number of nodes at level i and the work per node at level i, so we get $2^i c_1$.

So our summation looks like

$$\sum_{i=0}^{\log n} 2^i c_1 = c_1 \sum_{i=0}^{\log n} 2^i$$

$$\sum_{i=0}^{\log n} 2^i c_1 = c_1 \sum_{i=0}^{\log n} 2^i$$

This expands to a term like:

$$c_1(1 + 2 + 4 + ... + n)$$

where we know the inner term to be in $O(n)$. So ultimately, our bound is $O(n)$. That's a logarithmic improvement over `inord`!

Case The **work** of `inord'` on an **unbalanced** tree.

```
fun inord' (Empty : tree, acc : int list) = acc
  | inord' (Node (L, x, R), acc) =
      inord' (L, x :: inord' (R, acc))
```

Then we would get:

$$W_{\texttt{inord'}}(0) = c_0$$

$$W_{\texttt{inord'}}(n) = W_{\texttt{inord'}}(n-1) + W_{\texttt{inord'}}(0) + c_1$$

By analogy, we've seen this recurrence before. This solves to

$$W_{\texttt{inord'}}(n) = W_{\texttt{inord'}}(n-1) + c_0 + c_1$$

which is in $O(n)$. So we do the same amount of work.

```
fun inord' (Empty : tree, acc : int list) = acc
  | inord' (Node (L, x, R), acc) =
      inord' (L, x :: inord' (R, acc))
```

Now finally, let's do the span analysis. Let's assume the best case, which is a balanced tree.

Case The **span** of `inord'` on a **balanced** tree.

$$S_{\texttt{inord'}}(0) = c_0$$

$$S_{\texttt{inord'}}(n) = 2 \cdot S_{\texttt{inord'}}(\frac{n}{2}) + c_1$$

```
fun inord' (Empty : tree, acc : int list) = acc
  | inord' (Node (L, x, R), acc) =
      inord' (L, x :: inord' (R, acc))
```

What gives? We still have two calls to $S_{\texttt{inord'}}\left(\frac{n}{2}\right)$, even though we usually get to take the max of them.

The reason is because there is a **data dependency** between the two calls to `inord'`. The call to `inord'(R, acc)` is being given as an argument to the other, meaning that the second call cannot be executed until the first finishes!

So our span bound ends up still being $O(n)$. This holds in the unbalanced case too.[7]

[7] Exercise, reader, etc etc.

Complexity of `inord`	Work	Span
Balanced	$O(n \log n)$	$O(n)$
Unbalanced	$O(n^2)$	$O(n^2)$

Complexity of `inord'`	Work	Span
Balanced	$O(n)$	$O(n)$
Unbalanced	$O(n)$	$O(n)$

4 - Sorting

We've now discussed trees and lists in detail. We've seen how we can analyze the performance of functions on these data structures, which cover a wide variety of classic computer science problems.

We will now turn to one of the most classic problems of all in computer science: sorting a list of integers.[8]

[8] Second only to fixing your parents' printer.

There are a variety of sorting algorithms that have been invented. We're going to try our hand at implementing a classic one – insertion sort.

Insertion sort works via repeatedly inserting an element into an already-sorted list. By doing this for every element in the list, we will eventually sort the entire list.

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

Insert Implementation

Let's implement the insertion function:

```
fun ins (x : int, [] : int list) : int list = [x]
  | ins (x, y::ys) =
      if x < y then
        x::y::ys
      else
        y :: ins (x, ys)
```

Now we can proceed to defining our sorting function!

`insort : int list -> int list`
REQUIRES: `true`
ENSURES: `insort L` is a sorted permutation of `L`

```
fun insort ([] : int list) : int list = []
  | insort (x::xs) = ins (x, insort xs)
```

How simple!

```
fun ins (x : int, [] : int list) : int list = [x]
  | ins (x, y::ys) =
      if x < y then
        x::y::ys
      else
        y :: ins (x, ys)
```

We see that insertion sort admits a very simple implementation in SML.

Now, let's analyze it!

Where n is the length of the list `L` in the expression `ins (x, L)`:

$$W_{\texttt{ins}}(0) = c_0$$

$$W_{\texttt{ins}}(n) = W_{\texttt{ins}}(n-1) + c_1 = ... = O(n)$$

```
fun insort ([] : int list) : int list = []
  | insort (x::xs) = ins (x, insort xs)
```

Now, if we analyze `insort`, we get:

Where n is the length of the list `L` in the expression `insort L`:

$$W_{\texttt{insort}}(0) = c_0$$

$$W_{\texttt{insort}}(n) = W_{\texttt{insort}}(n-1) + W_{\texttt{ins}}(n-1) + c_1$$

where the second equation is because the length of `insort xs` is $n-1$, since it's the same length as `xs`.

Now we solve:

$$
\begin{aligned}
&= W_{\texttt{insort}}(n) \\
&= W_{\texttt{insort}}(n-1) + W_{\texttt{ins}}(n-1) + c_1 \\
&= W_{\texttt{insort}}(n-1) + c_2 \cdot (n-1) + c_1 \\
&= W_{\texttt{insort}}(n-2) + c_2 \cdot (n-2) + c_1 + c_2 \cdot (n-1) + c_1 \\
&= \ldots \\
&= c_2 \cdot (1 + 2 + 3 + \ldots + (n-1)) + c_1 \cdot n \\
&= O(n^2)
\end{aligned}
$$

So insertion sort is quadratic time, which is expected.

Unfortunately, there is no real span analysis to be had here. On a list, the amount of opportunities for parallelism is low.

This might mean our dreams of analyzing the span of a sorting algorithm are dead!

Fortunately, someone else invented merge sort.[9]

Def **Merge sort** is a sorting algorithm involving dividing the list to be sorted in half, and recursively sorting each half.

Not only does merge sort achieve a better sequential complexity, but we will see how its span bound improves as well.

[9] Well, quick sort too. But we will discuss merge sort for today.

Our algorithm will be as follows:

- Split the list into two halves. It doesn't really matter how.
- Recursively sort either half.
- Merge the two sorted halves to make a sorted list.

The main important thing here is that it is pretty easy to split a list in half, as well as put two sorted lists together into another sorted list.

We will implement this, and call those functions `split` and `merge`.

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

REQUIRES: `true`

ENSURES: `split L` $\hookrightarrow$ `(A, B)` such that `L` is a permutation of `A @ B`, and `A` and `B` are roughly the same length

```
fun split ([] : int list) : int list * int list = []
  | split [x] = [x]
  | split (x::y::xs) =
      let
        val (A, B) = split xs
      in
        (x::A, y::B)
      end
```

```
merge : int list * int list -> int list
```

REQUIRES: `L` and `R` are sorted

ENSURES: `merge (L, R)` is a sorted permutation of `L @ R`

```
fun merge ([] : int list, R : int list) : int list = R
  | merge (L, []) = L
  | merge (x::xs, y::ys) =
      if x < y then
        x :: merge (xs, y::ys)
      else
        y :: merge (x::xs, ys)
```

Now that we've defined `split` and `merge`, we're ready to write `msort`.

`msort : int list -> int list`
REQUIRES: `true`
ENSURES: `msort L` evaluates to a sorted permutation of `L`

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

This code should almost read like pseudocode to you. In isolation, the `split` and `merge` functions do precisely what they were supposed to do, and take away a great deal of the cognitive effort in understanding what the `msort` function does.

`msort` does as promised – splits the list, recursively sorts the halves, and then merges them together. There's very little extra fat to the logic.

Note We need the singleton case for `msort`, because otherwise `split` will produce another singleton, which we will call `msort` on, which is an infinite loop.

```
fun split ([] : int list) : int list * int list = []
  | split [x] = [x]
  | split (x::y::xs) =
      let
        val (A, B) = split xs
      in
        (x::A, y::B)
      end

fun merge ([] : int list, R : int list) : int list = R
  | merge (L, []) = L
  | merge (x::xs, y::ys) =
      if x < y then
        x :: merge (xs, y::ys)
      else
        y :: merge (x::xs, ys)

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

That's all!

Now, let's analyze the complexity of `msort`. We'll do the work first, and then the span.

Note We will assume, but not show, that the complexity of `split` and `merge` are linear in the sizes of their inputs.

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

Where n is the length of the list `L` in the expression `msort L`:

$$W_{\texttt{msort}}(0) = c_0$$

$$W_{\texttt{msort}}(1) = c_1$$

$$W_{\texttt{msort}}(n) = 2 \cdot W_{\texttt{msort}}\left(\frac{n}{2}\right) + W_{\texttt{split}}(n) + W_{\texttt{merge}}\left(\frac{n}{2}, \frac{n}{2}\right) + c_2$$

$$W_{\texttt{msort}}(n) = 2 \cdot W_{\texttt{msort}}\left(\frac{n}{2}\right) + W_{\texttt{split}}(n) + W_{\texttt{merge}}\left(\frac{n}{2}, \frac{n}{2}\right) + c_2$$ [10]

Now we can solve to:

$$\begin{aligned} &= W_{\texttt{msort}}(n) \\ &= 2 \cdot W_{\texttt{msort}}\left(\frac{n}{2}\right) + W_{\texttt{split}}(n) + W_{\texttt{merge}}\left(\frac{n}{2}, \frac{n}{2}\right) + c_2 \\ &= 2 \cdot W_{\texttt{msort}}\left(\frac{n}{2}\right) + n \cdot c_3 + c_2 \end{aligned}$$

It turns out that this is the same as another recurrence we saw earlier, the balanced work recurrence for `inord`, because we have two calls at size $\frac{n}{2}$, and linear work at each node. This solves to $O(n \log n)$.

[10] Here, we use the notation $W_{\texttt{merge}}(\frac{n}{2}, \frac{n}{2})$ because the work of `merge` actually depends on both of its arguments. In this case, however, it still ends up just being $n \cdot c_3$, though (combined with the work from `split`)

What about span? `msort` makes two calls to itself in parallel, so there is an opportunity for a speedup.

We note that the span of `merge` and `split` must be the same as the work, though we don't show that here.

Where n is the length of the list `L` in the expression `msort L`:[11]

$$\begin{aligned}
&S_{\texttt{msort}}(n) \\
&= \max\left(S_{\texttt{msort}}\left(\frac{n}{2}\right), S_{\texttt{msort}}\left(\frac{n}{2}\right)\right) + S_{\texttt{split}}(n) + S_{\texttt{merge}}\left(\frac{n}{2}, \frac{n}{2}\right) + c_2 \\
&= S_{\texttt{msort}}\left(\frac{n}{2}\right) + S_{\texttt{split}}(n) + S_{\texttt{merge}}\left(\frac{n}{2}, \frac{n}{2}\right) + c_2 \\
&= S_{\texttt{msort}}\left(\frac{n}{2}\right) + n \cdot c_2 + c_3
\end{aligned}$$

[11] Base cases same as work.

$$S_{\texttt{msort}}(n) = S_{\texttt{msort}}\left(\frac{n}{2}\right) + n \cdot c_2 + c_3$$

Now we solve:

$$\begin{aligned}
&= S_{\texttt{msort}}(n) \\
&= S_{\texttt{msort}}\left(\frac{n}{2}\right) + n \cdot c_2 + c_3 \\
&= S_{\texttt{msort}}\left(\frac{n}{4}\right) + \frac{n}{2} \cdot c_2 + n \cdot c_2 + c_3 \\
&= \ldots \\
&= (1 + 2 + 4 + \ldots + n) \cdot c_2
\end{aligned}$$

So we get that, in parallel, merge sort is in $O(n)$.

That's pretty huge! The power of parallelism offers us to not just get a speedup when doing computations, but mathematically prove that we achieve a better asymptotic bound – a linear time sort. That's pretty cool.

There was a lot this lecture. Here's the highlights:

- We can analyze the work/span of a function on trees in terms of its depth d
- We can use the tree method to solve recurrences that make 2 or more recursive calls, by summing the cost of each level of the call tree
- We analyzed `inord` in four cases, and found that `inord'` beat it in all respects
- We implemented insertion and merge sorting algorithms extremely tersely
- We found that merge sort could be parallelized

Thank you!