Lesson 6
ASYMPTOTIC ANALYSIS

June 1, 2023

Last time, we learned about **trees**, which are a particular instance of **datatype declarations**.

We saw how we could represent trees via the `Empty` and `Node` constructors, as well as how to do proof by structural induction on a tree.

We then saw how we could use `datatype` declarations more generally, to define types and values that fit the problem we are trying to solve. We used it to define an `order` datatype, for comparing two integers.

1 - Asymptotic Complexity

So far, we've been concerned with exploring the expressivity of the Standard ML language.

We've learned how we can use fundamental language concepts like pattern matching, datatype declarations, and recursion to both state and solve various problems in computer science.

On the first day, functional programming was first explained as a kind of better way of communicating, as programmers. This is not the only thing we care about, however! We also care about how well our code performs.

Performance can be difficult to measure, however, because it's not a very standardized process! Performance can vary from program to program, of course, but also from computer to computer, depending on the hardware.

Even worse is that even on the same computer, performance can vary from run to run, due to factors in the computer such as its current CPU load, the amount of power it has, and other low-level factors.

We want to be able to analyze performance mathematically, in a way that is agnostic to these silly details.

We also observe that as time goes on, the amount of data that computers are asked to deal with has gone up significantly. Gone are the days of punching data into cards, now we store petabytes of data in the cloud.

As such, we want to measure performance in a way agnostic to silly hardware details, but still sensitive to mathematical differences in how an algorithm runs.

Def We use **Big-O notation** to describe the behavior of a function, in the limit of the size of its input. This function usually denotes the computational cost of a program.

Formally, we take a function $f : \mathbb{N}^+ \to \mathbb{N}^+$, which in our case describes the abstract cost of running some program.

The input is some metric of the data input (the length of a list, the size of a number) and the output is the number of abstract units that it costs to run on that input.

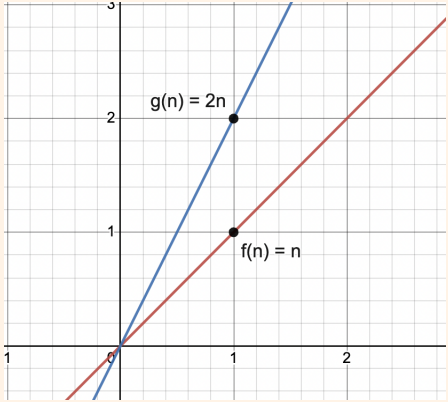
For another function $g : \mathbb{N}^+ \to \mathbb{N}^+$, we say that $f \in O(g)$ if there exists $n_0, c > 0$ such that, for all $n \geq n_0$, $f(n) \leq cg(n)$.

OK, that's a lot to digest. What does it really mean?

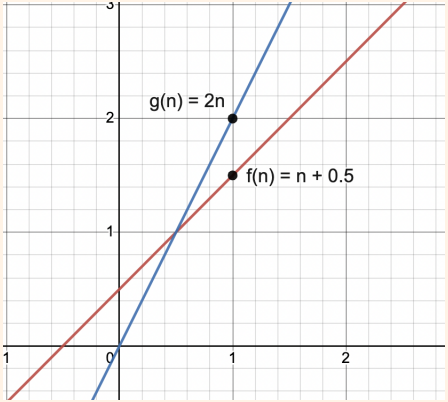
All this is describing is that a function f (the run-time function) is in $O(g)$, such as $O(n^2)$ or $O(n)$ (the complexity class), if there exists a a point beyond which the run-time function is *always* less than the complexity class function (scaled by a constant factor).

We need the constant factor for cases such as $f(n) = 2n$ and $g(n) = n$, where there is no value such that $f(n) \leq g(n)$. Despite that, however, the former function is still a linear function, so we should be able to scale g by a constant factor (say, 2.5) and conclude $f \in O(g)$.

This is why n and n_0 are important:

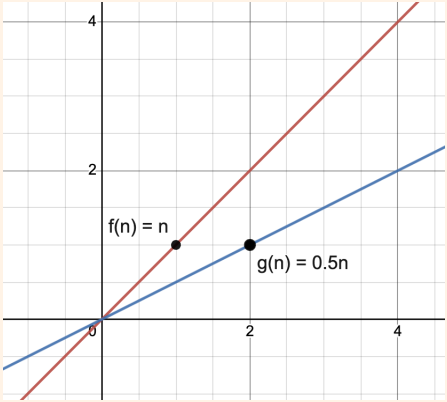


f is always less than g.

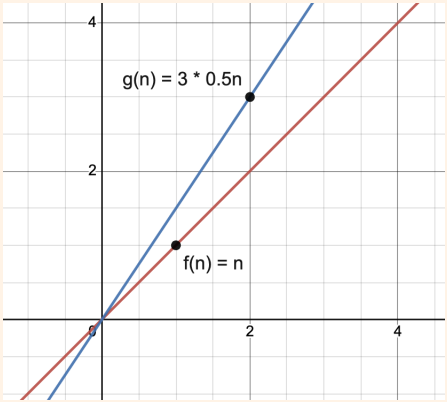


f is not always less than g

This is why c is important:



f is greater than g



but not when g is scaled by 3

In general, in this class, time complexity will fall into a few common buckets, and not really venture outside of them.

It is important to know how these complexity classes are related!

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < ...$$

In general, we will only see complexity up until $O(n^2)$.

It is worth noting that the $\in$ relation for asymptotic complexity is more permissive than we need! If we have $f \in O(n)$, then it is also true that $f \in O(n \log n)$, and $f \in O(n^2)$, and so on.

We want to find the *least* complexity class that captures this relation. This is called a *tight asymptotic bound*.

Here are some examples of common operations that fall into these buckets:

- $O(1)$ - Consing an element onto a list, multiplying two numbers, stepping an expression once
- $O(\log n)$ - Binary searching an interval n wide, finding an element in a binary search tree
- $O(n)$ - Computing the length of a list, finding the last element of a list, summing the nodes of a tree
- $O(n \log n)$ - Mergesort and quicksort
- $O(n^2)$ - Insertion sort, selection sort

Note that, when computing asymptotic complexity, there is always an underlying function f that we are approximating the behavior of. This function always takes in some number that denotes the "size of the input".

In the case of numbers, this is just the number itself.

In the case of something like "computing the length of the list", this is the length of the list. In the case of trees, this could be the number of nodes in the tree.

It's important to realize that, although we usually arecasual and don't specify things explicitly, these underlying size metrics are still there! We will be explicit about it going forward.

2 - Work and Recurrences

Big-O notation is useful conceptually, but not very useful if we can't reliably come to define the function f that we would like to find the bound of!

Casually, we might skip the function f and just go straight to the bound, by estimating the amount of times that work is done. For instance, for the following function:

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length xs
```

without even thinking, we might say that it is $O(n)$, because we iterate over the entire list.

This is an informal way of thinking, however! One of the strengths of functional programming is that we will be able to give this analysis a more rigorous treatment.

To do this, we will specify a **recurrence relation** for each function that we wish to analyze.

Def A **recurrence relation** is a series of mathematical equations that specifies the run-time cost of a recursive function, defined in terms of itself.

We then solve the recurrence relation to obtain a function that describes the **work** of the function, in some size metric, and then place it in a complexity class.

Def The **work** of a function is its run-time cost.

Let's take the `fact` function, as a starter.

```
fun fact (0 : int) : int = 1
  | fact n = n * fact (n - 1)
```

We obtain the recurrence by analyzing the work done in each case. We obtain two equations, for the two cases of the function:

$$W_{\texttt{fact}}(0)^1 = c_0$$
$$W_{\texttt{fact}}(n) = c_1 + W_{\texttt{fact}}(n-1)$$

[1] $W_{\texttt{fact}}(0)$, for "work of `fact` on input size 0"

$$W_{\texttt{fact}}(0) = c_0$$
$$W_{\texttt{fact}}(n) = W_{\texttt{fact}}(n-1) + c_1$$

In the first case, we return an unspecified constant c_0. Work is measured in arbitrary abstract units, and we don't necessarily know how many of those are taken up by returning 1. It's nonzero, because there is some amount of work that needs to be done, but it's otherwise unknown, so we use a placeholder constant c_0.

In the second case, we do some constant amount of work, again. It's not necessarily the exact same as the previous case, so we just call it c_1. We also do the work of computing `fact` on an input that is 1 smaller.

Now that we have the recurrence, we can go ahead and try to solve. We will use a technique called **unrolling**.

Def The **unrolling** method for solving recurrences entails just expanding the definition of the recurrence, finding a pattern, and then writing a closed form.

So we obtain, when n is the size of the number `n` in the expression `fact n`:

$$\begin{aligned} W_{\texttt{fact}}(n) &= W_{\texttt{fact}}(n-1) + c_1 \\ &= W_{\texttt{fact}}(n-2) + c_1 + c_1 \\ &= \dots \\ &= \sum_{i=0}^{n} c_1 + c_0 \\ &= n \cdot c_1 + c_0 \end{aligned}$$

When using the unrolling method, we're not being super strict, but the idea is just to demonstrate that you know that there is a pattern, and that the recurrence definitely will expand to the given closed form!

We are not going to be super picky, but it matters that we can follow along with your reasoning.

Finally, once we have a closed form, this is essentially our function f. We now want to find a complexity class for it.

If you followed all the steps correctly, it should be fairly obvious what is a tight bound for it. For instance, for the closed form

$$W_{\texttt{fact}}(n) = n \cdot c_1 + c_0$$

we can clearly tell that it is in $O(n)$, because it is a linear function in n.

Consider the following artificial SML function:

```
fun findFirstEven ([] : int list) : int option = NONE
  | findFirstEven (x::xs) =
      if x mod 2 = 0 then
        SOME x
      else
        findEven xs
```

This function, `findFirstEven`, seeks to find the first even number in an SML list of integers. Since there might not be such a number, it returns an optional value.

How might we write a recurrence for this function?

First, we need to identify the parameter for our recurrence. What number is our recurrence measured in terms of?

For lists, we'll choose to pick the length of the list. It's the most obvious metric.

For the base case, it's fairly simple. We do a constant amount of work, so we produce:

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

What about the recursive case? Well, the recursive case is actually *two* cases.

Asymptotic complexity is measured in terms of **worst-case behavior**.

This means that, across the entire range of inputs that the function could be given, we are interested in measuring the complexity when the function is given an input that causes the *most work for it to do*.

While `findFirstEven`, in the recursive case, *could* terminate immediately upon looking at the first number, to be truly pessimistic we will assume that there is no such even number in the list, which will cause us to always enter back into the recursive case.

When n is the length of `L` in the expression `findFirstEven L`:

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

because we call the function again on a list of length one smaller, and do a constant amount of work.

Now that we've done a few examples, we're ready to identify the general formula for analyzing the asymptotic complexity of a function:

Given an SML function `f` that we seek to find the complexity of, we should:

- Identify the size parameter, n, that the recurrence is in terms of. This could be something like the length of a list, the size of the number, etc.
- Write a recurrence for the function. This entails writing an equation for every case it takes, in the *worst case*.
- Simplify the recurrence into a closed form in n.
- Estimate a complexity class from the recurrence

Remember the `rev` function?

```
fun rev ([] : int list) : int list = []
  | rev (x::xs) = rev xs @ [x]
```

Well, this is them now:[2]

```
fun trev ([] : int list, acc : int list) = acc
  | trev (x::xs, acc) = trev (xs, x::acc)
fun rev (L : int list) : int list = trev (L, [])
```

We said that our original implementation of `rev` was undesirable because it wasn't tail recursive.

Not only does that matter for how much space it takes, but how much time as well! Let's write the recurrence.

[2] Feel old yet?

```
fun @ ([] : int list, ys : int list) : int list = ys
  | @ (x::xs, ys) = x :: (xs @ ys)

fun rev ([] : int list) : int list = []
  | rev (x::xs) = rev xs @ [x]
```

But, we see that `rev` actually depends on `@`.

We need to first write a recurrence for `@`, before we can tackle `rev`.

But, how should we write a recurrence for `@`? It has two arguments! Our recurrences only take in a single size parameter.

We will just identify a single parameter that it makes sense to measure the size of the input in. We mentioned before that `@` only ever looks at the left list, so we'll only measure our work in terms of the length of the left list.

Where n is the length of `L`, when calling `L @ R`:

$$W_{@}(0) = c_0$$

$$W_{@}(n) = W_{@}(n) + c_1$$

We can write out the work as before, but we know that this will go to the closed form

$$n \cdot c_1 + c_0$$

which is in $O(n)$.

Now we write the recurrence for `rev`:

```
fun rev ([] : int list) : int list = []
  | rev (x::xs) = rev xs @ [x]
```

Where n is the length of `L`, in the expression `rev L`:

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

$$W_{\texttt{rev}}(n) = W_{\texttt{rev}}(n-1) + ???$$

We make a call to `rev`, which has work $W_{\texttt{rev}}(n-1)$. But what about after? We call `@`, but how big is the length of the list?
Fortunately, we know that the length of `rev xs` is the same as the length of `xs`, so we can write:

$$W_{\texttt{rev}}(n) = W_{\texttt{rev}}(n-1) + W_{\texttt{@}}(n-1) + c_1$$ [3]

[3] Don't forget this c_1. There is still a constant amount of work being done here.

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

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

Now to solve the recurrence, we get:

$$\begin{aligned}
&= W_{\texttt{rev}}(n) \\
&= W_{\texttt{rev}}(n-1) + W_{@}(n-1) + c_1 \\
&= W_{\texttt{rev}}(n-1) + c_2 \cdot (n-1) + c_1 \\
&= W_{\texttt{rev}}(n-2) + W_{@}(n-2) + c_1 + c_2 \cdot (n-1) + c_1 \\
&= W_{\texttt{rev}}(n-2) + c_2 \cdot (n-2) + c_1 + c_2 \cdot (n-1) + c_1
\end{aligned}$$

$$
\begin{aligned}
&= W_{\texttt{rev}}(n-2) + c_2 \cdot (n-2) + c_1 + c_2 \cdot (n-1) + c_1 \\
&= ... \\
&= \sum_{i=0}^{n} c_2 \cdot (n-i) + n \cdot c_1 + c_0
\end{aligned}
$$

The first term follows from adding a $c_2 \cdot (n-i)$ at the ith step of the expansion.
The second follows from adding a c_1 per step of the expansion.
The final term comes from the base case.
We apply a handy math fact, which is that $\sum_{i=0}^{n}(n-i) = \sum_{i=0}^{n} i = 1 + 2 + ... + n = \frac{n(n+1)}{2}$, which is upper bounded by n^2. So:

$$= O(n^2) \cdot c_2 + n \cdot c_1 + c_0 \in O(n^2)$$

`rev` is quadratic time, which is really bad!

In hindsight, we could have predicted this. The recursive case of `rev` essentially appends the prefix of the list over and over again, recursively.

But with a recurrence, we now have a mathematical justification for our asymptotic analysis. We know for sure that the `rev` function is quadratic time. What will we do about it?

Well, let's analyze `trev`. How much better does it do?

```
fun trev ([] : int list, acc : int list) = acc
  | trev (x::xs, acc) = trev (xs, x::acc)
```

To analyze `trev`, we note that the right list is never touched, so we will again measure in terms of the length of the first list.

Where n is the length of `L` in the expression `trev (L, acc)`:

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

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

We know what this solves to. We get a happy bound of $O(n)$.

We just demonstrated how we can use math and **recurrences** to formally reason about the run-time of functional programs.

This came out of the fact that due to **purity**, functions always behave the same on the same arguments. We can define our W function and know that it is truly a *function*, and returns the same results on the same inputs.

In addition, due to the recursive nature of the functions we wrote, the recurrences came naturally. It only took two cases to specify the run-time of the function, for most of our analyses, and solving it was just math.

Functional programs help us reason about our code!

Any questions?

Ask anonymously: menti.com with code 5840 8607

3 - Parallelism and Span

Earlier this lecture, we explored the idea of **work**, which is the run-time cost of a particular function, varied over the size of its input. We saw that we could measure work in terms of *abstract units*, related to the number of steps that we took to evaluate an expression, in our model of SML. For instance:

`(1 + 2) + (3 + 4)` $\Longrightarrow$ `3 + (3 + 4)` $\Longrightarrow$ `3 + 7` $\Longrightarrow$ `10`

We don't know how much cost a single step is, but we might say that it takes 3 steps to evaluate, by the above trace.

But this doesn't always need to be true.

Let's have a race!

In lecture, we're going to have one person count the number of people in the room.

I will race them to see who can count the number of people in the room first.

Let's see who wins.

Counting is a problem that can make use of **parallelism**.

Def **Parallelism** is when a process is able to execute some of its tasks at the same time.

Usually, this happens at the hardware level on a multicore machine, by the processor being able to delegate distinct instructions to be performed on different "cores", which can communicate answers with each other.

We will safely ignore most of that, and just assume that multicore machines exist, though.

It's hard to pick a number for how many, however. So let's just assume we have infinitely many.

Having infinite cores sounds like a superpower that should elevate us to computational deities. Infinite cores? Infinite power!

It's not quite the case, though.

Consider the problem of making a sandwich.

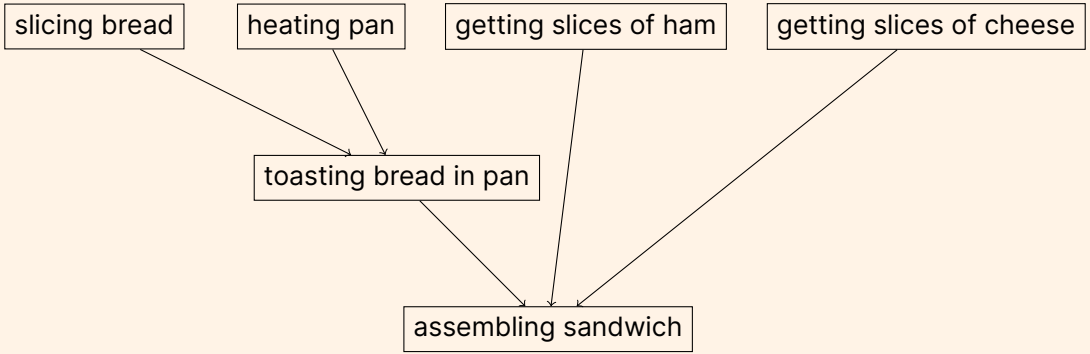
If there were an infinite amount of me, I might want infinitely many sandwiches.

But, I can't just instantly make a sandwich, even if I had more hands available. I have to toast the bread, which means I need to wait for the pan to heat up, and I can't toast the bread, because I need to slice it from the loaf, and I can't slice it from the loaf, because there's only two sides and maybe another version of me wants to slice it first.

Note Making a sandwich is hard.

This is the notion of **task dependency** in parallel computing.

Def When computing in parallel, some tasks have **dependencies**, meaning they cannot be completed until other tasks are finished.



Question: **How long does it take to make a sandwich?**

Answer: **The length of the longest path.**

This demonstrates the notion of a **task dependency graph**.

Def A **task dependency graph** is an acyclic graph which shows the relationship between tasks and their dependencies. The nodes are tasks, annotated with time to complete, and the edges go from tasks that are preconditions to other tasks.

In a task dependency graph, with only a single processor, we cannot avoid having to visit every single node. It doesn't really matter what order we do it in, but we have to pay cost equal to the sum of all the nodes.

With infinitely many processors, because we can start unrelated tasks at the same time, the amount of time spent is just the length of the *longest* dependency chain in the graph. In essence, the height of the graph.

In terms of concrete vocabulary, this demonstrates the difference between **work** and **span**.

Def We say that the **work** of a process is the time expended using a single processor.

Def We say that the **span** of a process is the time expended using infinitely many processors.

We said earlier that in SML, tuples are evaluated left to right. In a world with infinitely many processors, we will assume that elements of a tuple can be computed *at the same time*.

We wrote recurrences labeled with W earlier – W, for *work*. Now, we will write recurrences for *span*.

Let's compute the span of the `length` function. The process will look very similar, but we will assume that any tuples are evaluated in parallel.

Note This is a general rule going forward! Whenever you see a tuple, when we are doing span computations, you may assume that the elements of the tuple are being evaluated **in parallel**. This is *only* for tuples.

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length xs
```

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

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

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

Then we have:

$$\begin{aligned} S_{\texttt{length}}(n) &= \max(c_1, S_{\texttt{length}}(n-1)) + c_2 \\ &= S_{\texttt{length}}(n-1) + c_2 \\ &= ... \\ &= O(n) \end{aligned}$$

What gives?

We got the same bound, even though we had infinitely many processors!

This is because lists are inherently a sequential structure. Even if you had many processors, you can't touch the second list element until you look at the first. In essence, every list has a single "child", if we view `xs` as the child to `x::xs`.

What's a data structure that has more than one child?

Let's analyze the span of a function on trees!

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

We need a metric for the size of the tree, though. There's actually two options – the number of nodes in the tree, or the height of the tree.

Both are valid ways to go about it. Let's try it first by using n, the number of nodes in the tree.

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

Where n is the number of nodes in `T`, in the expression `treesum T`:

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

What should we put for the recursive case? Because of infinite processors, we can consider the calls to `treesum L` and `treesum R` to all be executed in parallel. But what is the number of nodes in each?

For an arbitrary tree, we can't possibly know the number of nodes in the left and right tree. This is where worst-case analysis will save us.

To simplify our analysis, we will simply pick the configuration that is the worst for us. The worst case for `treesum` is when the input tree is a "spine", or a straight line. Thus, we will assume the number of nodes in the left is $n-1$, and in the right, 0.

So we obtain:

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

$$S_{\texttt{treesum}}(n) = S_{\texttt{treesum}}(n-1) + c_1$$

where the constant work is again the residual work of addition and retrieving the other results.

$$S_{\texttt{treesum}}(0) = c_0$$
$$S_{\texttt{treesum}}(n) = S_{\texttt{treesum}}(n-1) + c_1$$

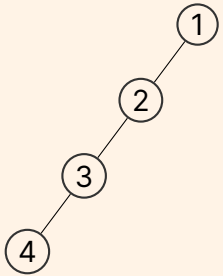
Wait a second, we've seen this before. If we swap the S for W, this is the same recurrence as we've been getting the whole time!

We know this is in $O(n)$.

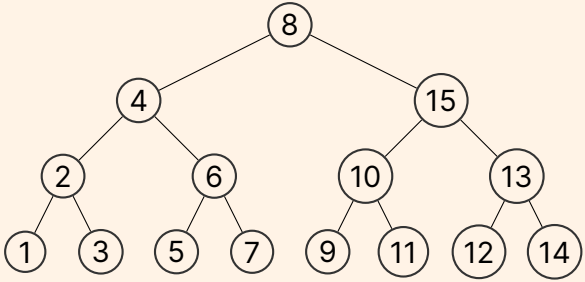
We see that our span bound on an unbalanced tree is still $O(n)$! Even with infinite processors, we can't do better than linear complexity.

This kind of makes sense, because a tree that is just a single line is basically a list with extra metadata. `treesum` isn't doing much more than `length`, so we get a similar span bound!

The main two cases we will be concerned about, for binary trees, will be the unbalanced and balanced cases.



An unbalanced tree, or "spine"



A balanced tree, or "complete tree"

The unbalanced tree is almost always the worst input for a given tree function.

Let's assume that the tree is balanced.

The base case is as before.

In this case, we can assume that each call to `treesum` is on a tree with roughly half the nodes as the input. For simplicity, let's assume we have a tree with $n = 2^k$, for some k.

Then, we get:

$$S_{\texttt{treesum}}(n) = \max(S_{\texttt{treesum}}(\frac{n}{2}), S_{\texttt{treesum}}(\frac{n}{2})) + c_1$$

$$S_{\texttt{treesum}}(n) = S_{\texttt{treesum}}(\frac{n}{2}) + c_1$$

$$
\begin{aligned}
&= S_{\texttt{treesum}}(n) \\
&= S_{\texttt{treesum}}(\frac{n}{2}) + c_1 \\
&= S_{\texttt{treesum}}(\frac{n}{4}) + c_1 + c_1 \\
&= S_{\texttt{treesum}}(\frac{n}{8}) + c_1 + c_1 \\
&= ... \\
&= \log n \cdot c_1
\end{aligned}
$$

because the number of times we can divide n by 2 is exactly the logarithm of n, base 2.

So this is in $O(\log n)$.

Finally, we obtain a better bound! On balanced trees, we can parallelize more work, so we are able to run `treesum` in only log time, as opposed to linear. That's a big difference!

We will see that parallelism is a powerful tool that will help us achieve better bounds in a variety of situations. This is only the tip of the iceberg.

One point that needs to be said before the end of lecture: yes, having infinite processors is impossible[4].

This doesn't cheapen the findings we discussed. We're dealing with mathematical abstractions, and being able to assume as many processors as needed is one of them.

In reality, performance is somewhere between the work and the span, but the span remains a useful mathematical ideal.

[4] I knew this the whole time!!!!

Thank you!