Multi-Pass Parallel

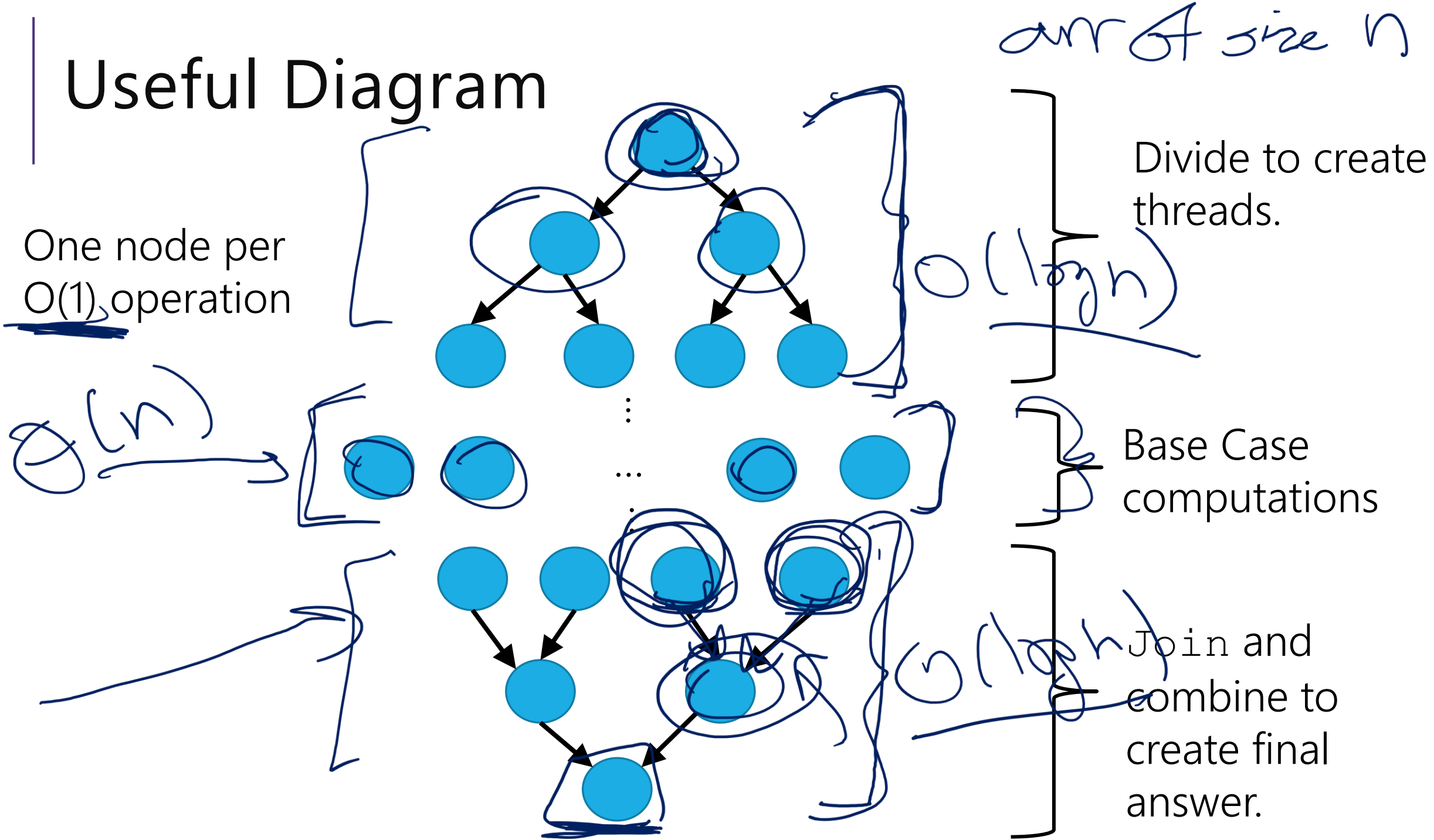
CSE 332 25Sp
Lecture 21

Announcements

	Monday	Tuesday	Wed	Thursday	Friday
This Week	TODAY Ex 9 (reductions, gs) due Ex 11 (parallel prog) out				Ex 10 (F-J prog) due Ex 12 (concurrency, GS) out
Next Week	Veteran's Day (no class)	Ex 11 due			Ex 12 due

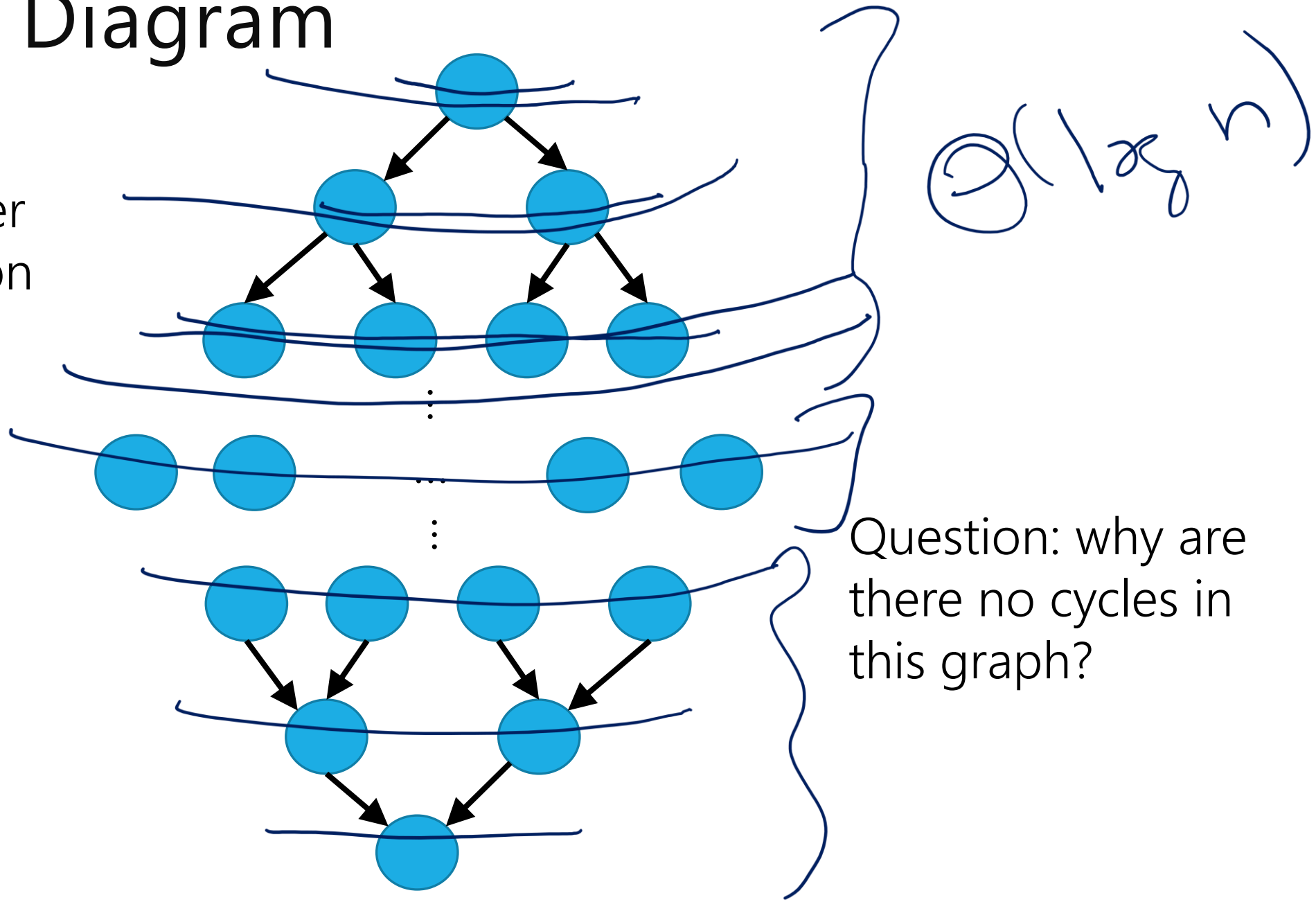
Optional readings (Grossman) covers next few weeks of parallelism and concurrency

Useful Diagram



Useful Diagram

One node per
O(1) operation



Question: why are there no cycles in this graph?

Analysis

Big idea: let P, the number of processors, be another variable in our big-O analysis.

Let T_P be the big-O running time with P processors.

What is T_P for summing an array?

$O\left(\frac{n}{P} + \log n\right)$

Definitions

Work: T_1

it's $O(n)$ for summing an array.

Probably (but not necessarily!!!) going to be equivalent to the running time of the code you would write if you had never heard of parallelism.

Definition is running time of **parallel** code if you had a single processor.

Span: T_∞

$O(\log n)$ for summing an array

$T_?$ is running time with ? Processors, so span is "you always have a processor available"

Longest path in graph of computation. "critical path"

More Definitions

Speedup: for P processors: $\frac{T_1}{T_P}$

ideally: speedup will be close to P ("perfect linear speedup")

E.g. if $T_1 = 100\text{sec}$ And $T_4 = 25\text{sec}$, then speedup$= \frac{100}{25} = 4$

Parallelism: $\frac{T_1}{T_\infty}$

the speedup when you have as many processors as you can use (there's a point at which another one won't actually help).

Optimal T_P

We can calculate T_1, T_∞ theoretically.

But we probably care about T_P for, say, $P = 4$.

T_P can't beat (make sure you understand why):

- T_1/P
- T_∞

$$\Omega\left(\frac{T_1}{P} + T_\infty\right)$$

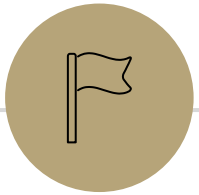
So optimal running time (asymptotically)

$$T_P = O((T_1/P)) + T_\infty)$$

ForkJoin Framework has **expected** time guarantee of that O()

Assuming you write your code well enough.

Uses randomized scheduling.



Amdahl’s Law

Amdahl's Law

Now it's time for some bad news.

In practice, your program won't **just** sum all the elements in an array.

You will have a program with

Some parts that parallelize well

- Can turn them into a map or a reduce.

Some parts that won't parallelize at all

- Operations on a linked list. (**data structures matter**!!!)
- Reading a text file.
- A computation where each step needs the result of the previous steps.

Amdahl's Law

Let the work be 1 unit of time.

Let S be the portion of the code that is unparallelizable ("sequential").

$T_1 = S + (1 - S) = 1.$

At best we can get perfect linear speedup on the parallel portion

$T_P \geq S + \frac{1-S}{P}$

So overall speedup with P processors

$$\frac{T_1}{T_P} \leq \frac{1}{S+(1-S)/P}$$

Therefore Parallelism: $\frac{T_1}{T_\infty} \leq \frac{1}{S}$

Amdahl's Law

Suppose our program takes 100 seconds.

And S is 1/3 (i.e. 33 seconds).

What is the running time with

3 processors

6 processors

22 processors

67 processors

1,000,000 processors (approximately).

Amdahl's Law

$$\frac{T_1}{T_P} \leq \frac{1}{S + \frac{1-S}{P}}$$

Amdahl's Law

Suppose our program takes 100 seconds.

And S is 1/3 (i.e. 33 seconds).

What is the running time with

3 processors: 33 + 67/3 ≈ 55 seconds

6 processors: 33 + 67/6 ≈ 44 seconds

22 processors: 33 + 67/22 ≈ 36 seconds

67 processors 33 + 67/67 ≈ 34 seconds

1,000,000 processors (approximately). ≈ 33 seconds

Amdahl's Law

$$\frac{T_1}{T_P} \leq \frac{1}{S + \frac{1-S}{P}}$$

Amdahl's Law

This is BAD NEWS

If 1/3 of our program can't be parallelized, we can't get a speedup better than 3.

No matter how many processors we throw at our problem.

And while the first few processors make a huge difference, the benefit diminishes quickly.

Amdahl's Law and Moore's Law

In the Moore's Law days, 12 years was long enough to get 100x speedup.

Suppose in 12 years, the clock speed is the same, but you have 256 processors.

What portion of your program can you hope to leave unparallelized?

$$100 \leq \frac{1}{S+\frac{1-S}{256}}$$

[wolframalpha says] $S \leq 0.0061$.

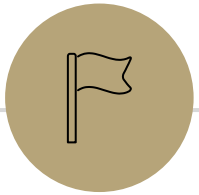
Amdahl’s Law and Moore’s Law

Moore’s Law was “a business decision”

- How much effort/money/employees are dedicated to improving processors so computers got faster.

Amdahl’s Law is a theorem

- You can prove it formally.



Parallel Prefix and Pack

Amdahl's Law: Moving Forward

Unparallelized code becomes a bottleneck quickly.

What do we do? Design smarter algorithms!

Consider the following problem:

Given an array of numbers, return an array with the "running sum"

Input:

3	7	6	2	4

Output:

3	10	16	18	22

Sequential Code

```
output[0] = input[0];
for(int i=1; i<arr.length;i++){
     output[i] = input[i] + output[i-1];
}
```

More clever algorithms

Doesn't look parallelizable…

But it is!

Algorithm was invented by Michael Fischer and Richard Ladner

- Both were in UW CSE's theory group at the time
- Richard Ladner is still around
  - Look for a cowboy hat…

For today: I'm not going to worry at all about constant factors.

Just try to get the ideas across.

Parallelizing

What do we need?

Need to quickly know the "left sum" i.e. the sum of all the elements to my left.

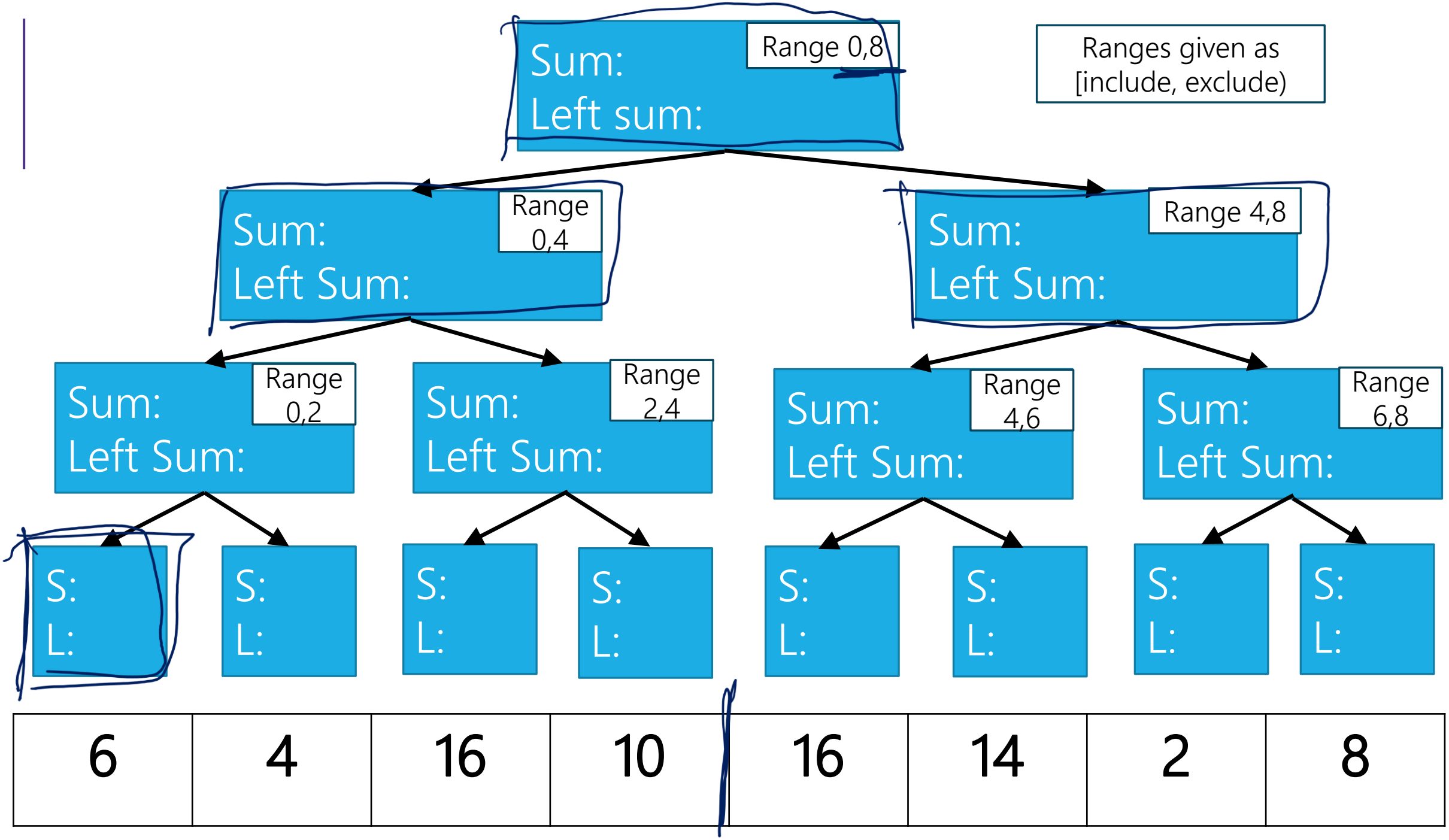
- in the sequential code that was `output[i-1]`

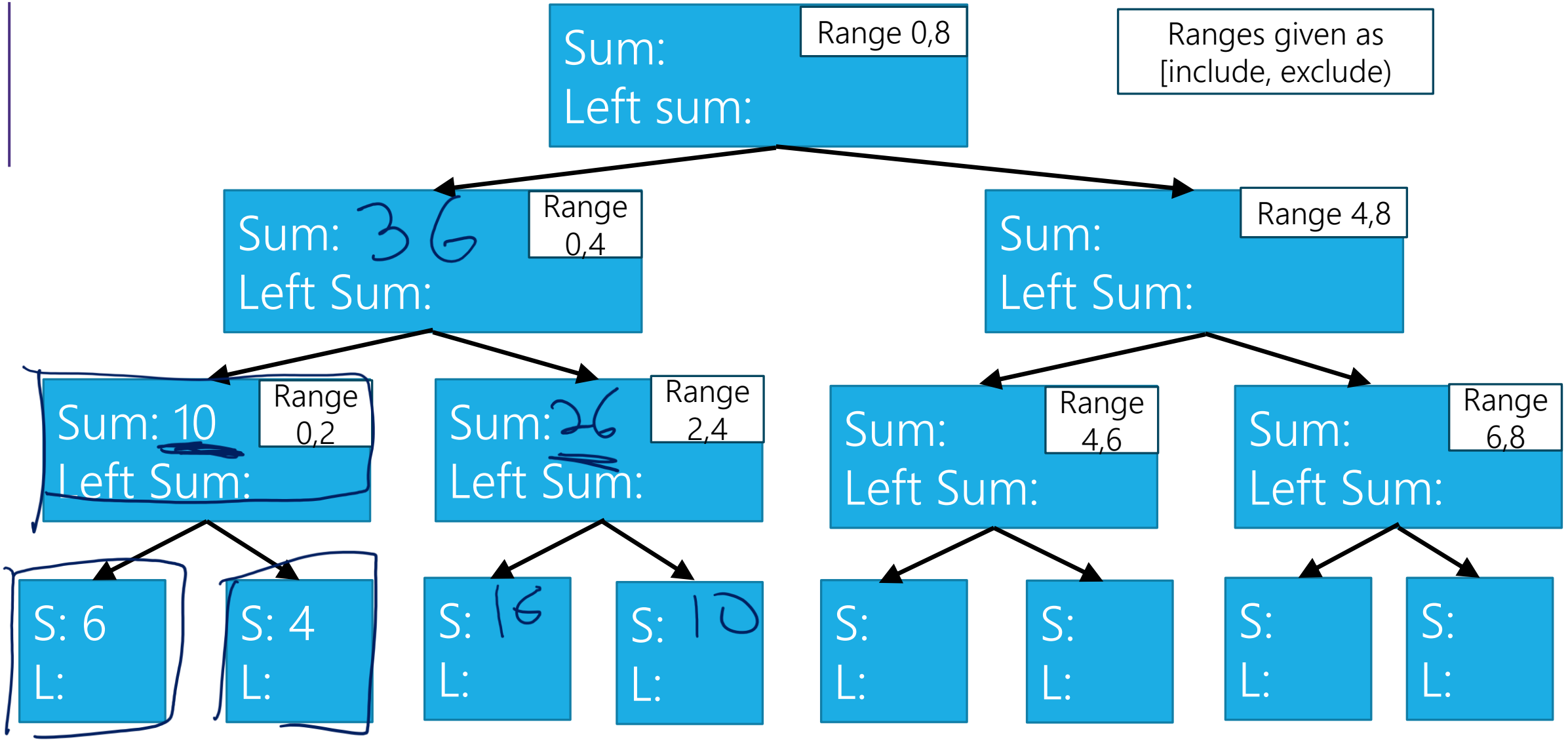
We'll use two passes,

The first sets up a data structure

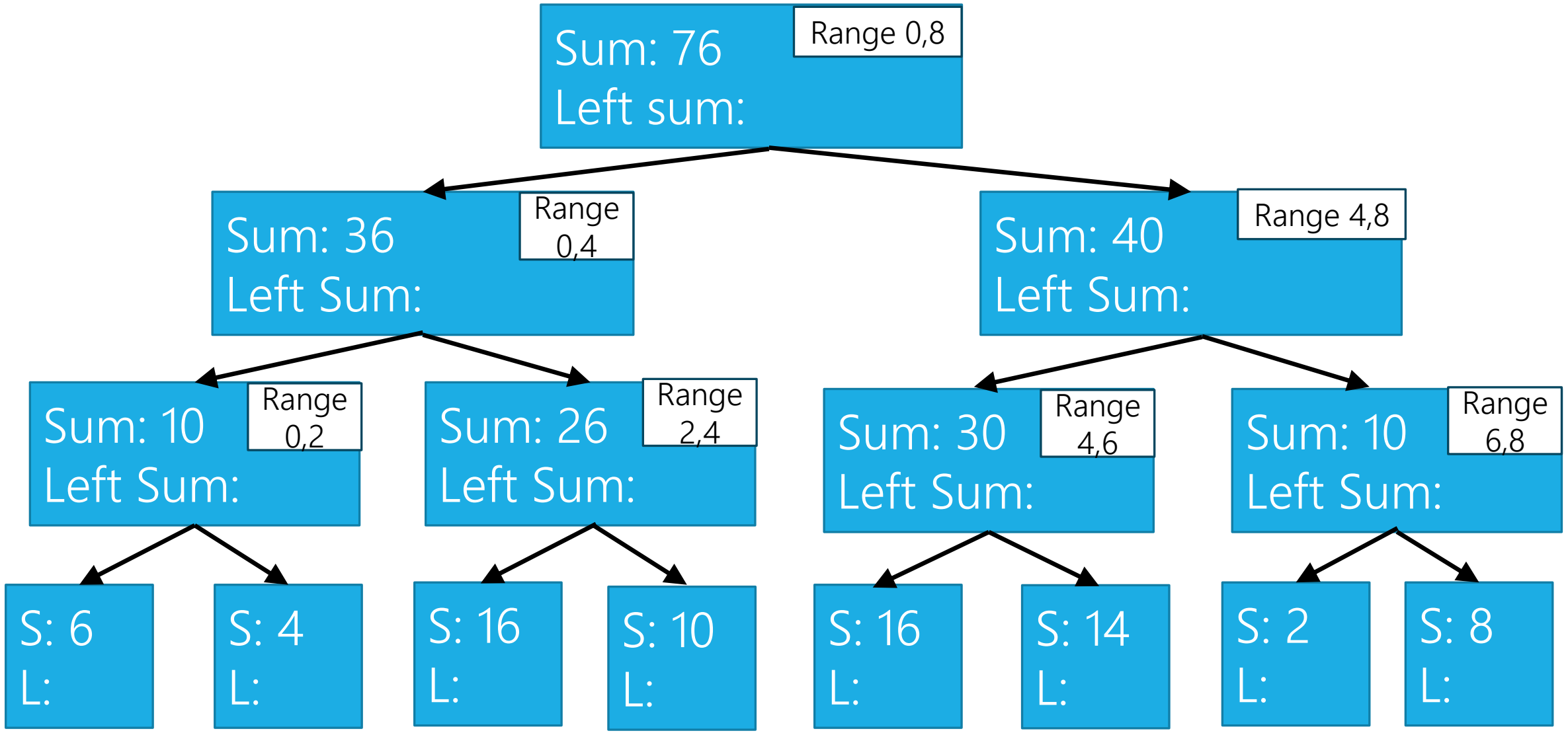
- Which will contain enough information to find left sums quickly

The second will assemble the left sum and finish the array.


Sum:
Left sum:
Range 0,8
Ranges given as
[include, exclude)
Sum:
Left Sum:
Range
0,4
Sum:
Left Sum:
Range 4,8
Sum:
Left Sum:
Range
0,2
Sum:
Left Sum:
Range
2,4
Sum:
Left Sum:
Range
4,6
Sum:
Left Sum:
Range
6,8
S:
L:
S:
L:
S:
L:
S:
L:
S:
L:
S:
L:
S:
L:
S:
L:
6
4
16
10
16
14
2
8



6	4	16	10	16	14	2	8



6	4	16	10	16	14	2	8

First Pass

Calculating those sums is the end of the first pass.

How long does it take in parallel?

Work:

Span:

Remember “work” is the running time on one processor.

“span” is the running time on infinitely many processors.

First Pass

Calculating those sums is the end of the first pass.

How long does it take in parallel?

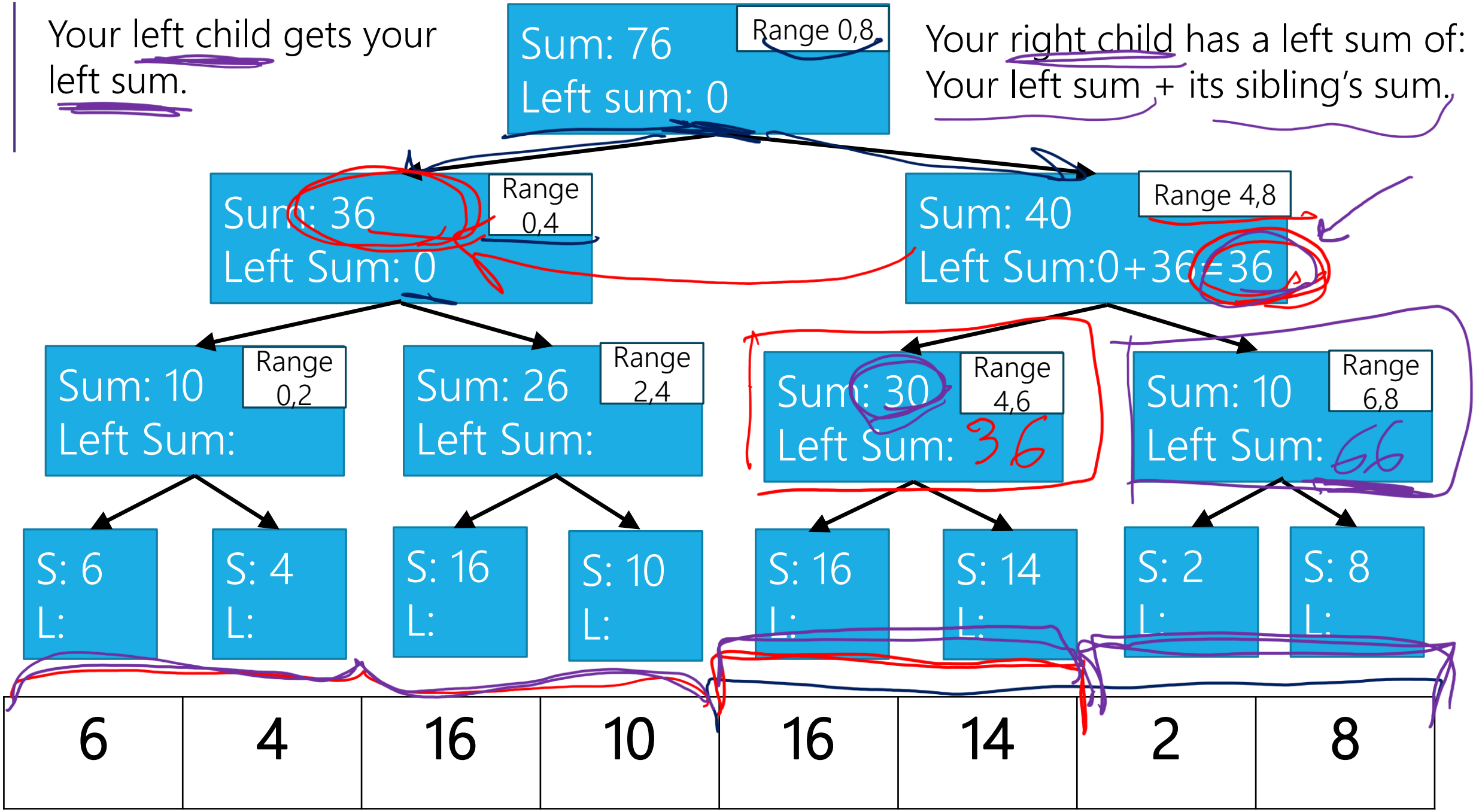
Work: $O(n)$

Span: $O(\log n)$

Just slightly modify our sum reduce code to build the data structure.

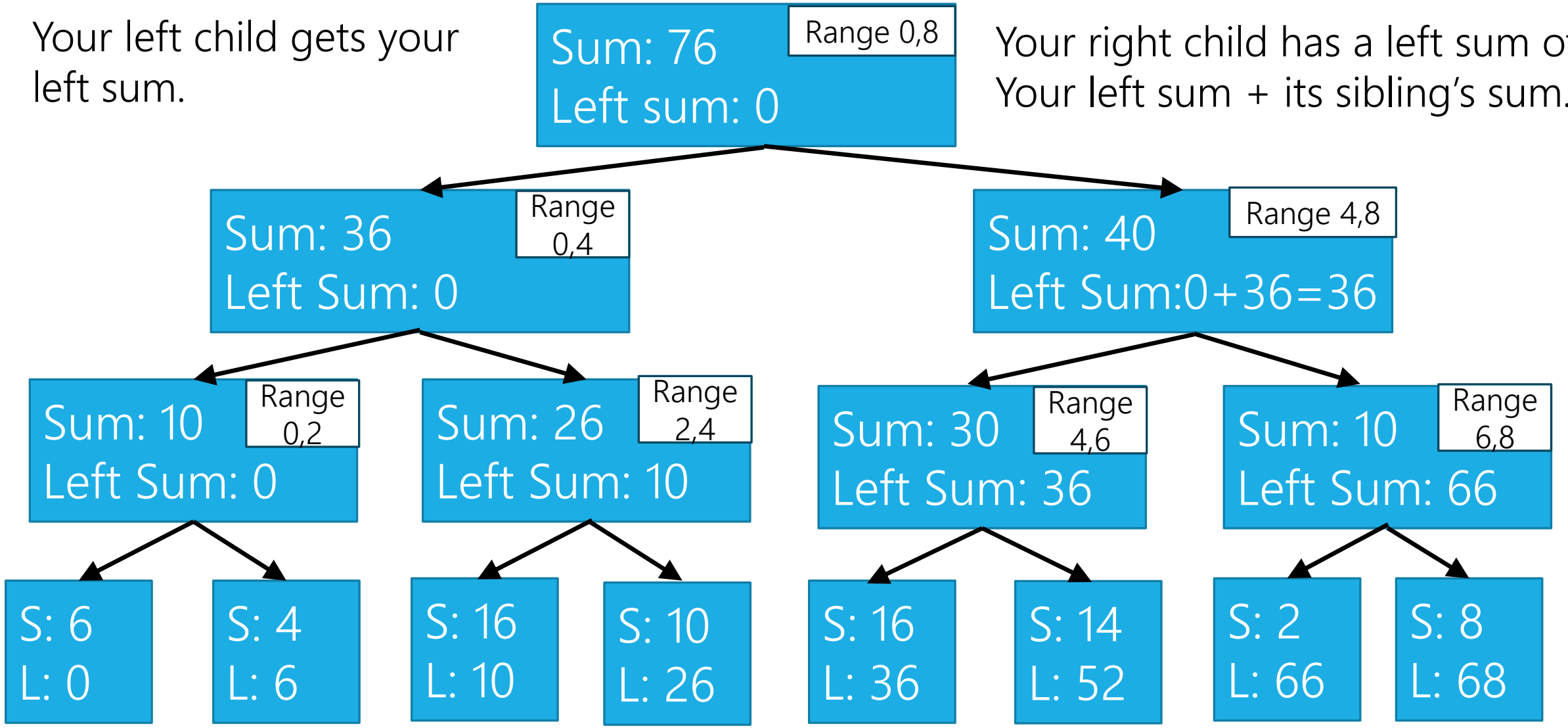
Your left child gets your left sum.

Your right child has a left sum of: Your left sum + its sibling's sum.



Your left child gets your left sum.

Your right child has a left sum of: Your left sum + its sibling's sum.



6	4	16	10	16	14	2	8

Second Pass

Once we've finished calculating the sums, we'll start on the left sums. Can we do that step in parallel?

YES!

Why are we doing two separate passes?
Those sum values have to be stored and ready.

Second pass has:
Work:

Span:

Second Pass

Once we've finished calculating the sums, we'll start on the left sums. Can we do that step in parallel?

YES!

Why are we doing two separate passes?
Those sum values have to be stored and ready.

Second pass has:
Work:$O(n)$

Span:$O(\log n)$

Third Pass

What's our final answer?

Our sequential code said element `i` of the new array should be

```
arr[i] + output[i-1]
```

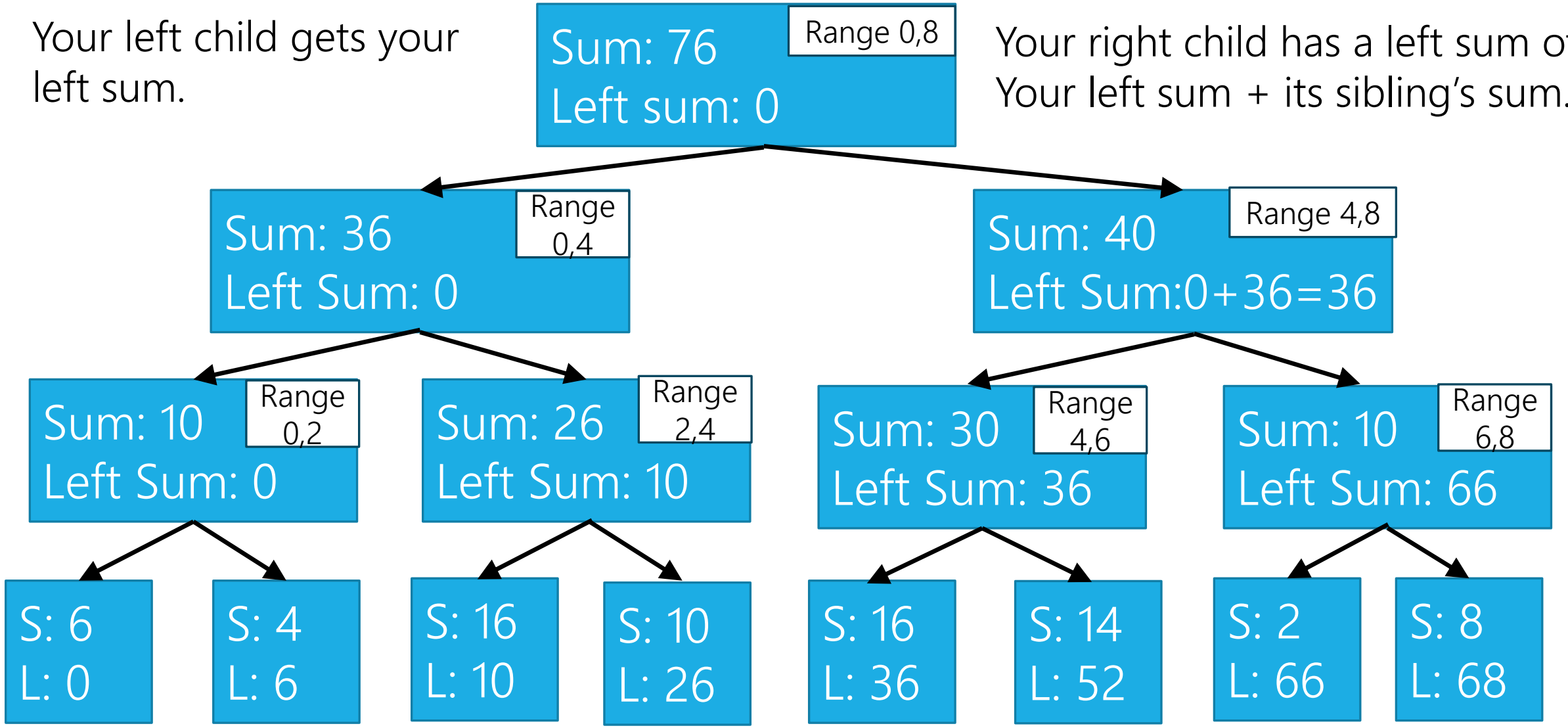
Or equivalently

```
arr[i] + left_sum[i]
```

Just need one more map using the data structure.

Your left child gets your left sum.

Your right child has a left sum of: Your left sum + its sibling's sum.



6	4	16	10	16	14	2	8
6	10	26	36	52	66	68	76

Analyzing Parallel Prefix

What's the

Work?

Span?

First pass was a slightly modified version of our sum reduce code.

Second pass had a similar structure

Third pass was a map

Analyzing Parallel Prefix

What's the

Work $O(n)$

Span $O(\log n)$

First pass was a slightly modified version of our sum reduce code.

Second pass had a similar structure.

Third pass was a map.

Our Patterns So Far

1. Map
   - Apply a function to every element of an array
2. Reduce
   - Create a single object to summarize an array (e.g., sum of all elements)
3. Prefix
   - Compute answer[i]=f(arr[i], answer[i-1])

Parallel Pack (aka Filter)

You want to find all the elements in an array meeting some property.

And move ONLY those into a new array.

Input:

6	4	16	10	16	14	2	8

Want every element >= 10

Output:

16	10	16	14

Parallel Pack

Easy – do a map to find the right elements…

Hard – How do you copy them over?

Parallel Pack

Easy – do a map to find the right elements…

Hard – How do you copy them over?

I need to know what array location to store in,

i.e. how many elements to my left will go in the new array.

Parallel Pack

Easy – do a map to find the right elements…

Hard – How do you copy them over?

I need to know what array location to store in,

i.e. how many elements to my left will go in the new array.

- Use Parallel Prefix!

Parallel Pack

Step 1: Parallel Map – produce bit vector of elements meeting property

6	4	16	10	16	2	14	8
0	0	1	1	1	0	1	0

Step 2: Parallel prefix sum on the bit vector

0	0	1	2	3	3	4	4

Step 3: Parallel map for output.

16	10	16	14

Step 3

How do we do step 3?

i.e. what's the map?

```
if(bits[i] == 1)
     output[ bitsum[i] - 1] = input[i];
```

Parallel Pack

We did 3 phases:

A map

A prefix

And another map.

Work:

Span:

Remark: You could fit this into 2 phases instead of 3. Won't change O().

Parallel Pack

We did 3 phases:

A map

A prefix

And another map.

Work: $O(n)$

Span: $O(\log n)$

Remark: You could fit this into 2 phases instead of 3. Won't change O().

Four Patterns

We've now seen four common patterns in parallel code

1. Map
2. Reduce
3. Prefix
4. Pack (a.k.a. Filter)

Making other code faster

Sometimes making parallel algorithms is just "can I turn my existing code into maps/reduces/prefixes/packs.

Other times parallel code with optimal span often requires changing to a different algorithm that parallelizes better.

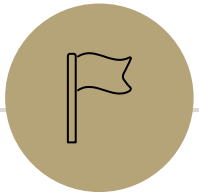
- These strategies often increase the work (slightly).

Two more optional examples: merge sort and quicksort, in parallel.

Details of the algorithms might change

- E.g., merge step in mergesort altered to run quicker in parallel.

Not responsible for them, but if you're curious, see these slides (or the Grossman text).



Parallel Sorts

Optional ☺

Parallelizing Quick Sort

```
Quicksort(){
    pick a pivot
    partition array such that:
        left side of the array is less than pivot
        pivot in middle
        right side of array is greater than pivot
    recursively sort left and right sides.
}
```

Quick Analysis Note

For all of our quick sort analysis, we'll do best case.

The average case is the same as best case.

Worst case is still going to be the same (bad) $\Theta(n^2)$ with parallelism or not.

Parallelizing Quick Sort

Step 1: Make the recursive calls forks instead of recursion.

What is the new (need some recurrences)

Work?

Span?

Parallelizing Quick Sort

Step 1: Make the recursive calls forks instead of recursion.

What is the new (need some recurrences)

Work? $T_1(n) = \begin{cases} 2T_1\left(\frac{n}{2}\right) + c_1 \cdot n & \text{if } n \geq \text{cutoff} \\ c_2 & \text{if } n < \text{cutoff} \end{cases}$

Span? $T_\infty(n) = \begin{cases} T_\infty\left(\frac{n}{2}\right) + c_1 \cdot n & \text{if } n \geq \text{cutoff} \\ c_2 & \text{if } n < \text{cutoff} \end{cases}$

Parallelizing Quick Sort

Step 1: Make the recursive calls forks instead of recursion.

What is the new (need some recurrences)

Work? $T_1(n) = \Theta(n \log n)$

Span? $T_\infty(n) = \Theta(n)$

Parallel Quick Sort

With infinitely many processors, we can speed up quicksort from $\Theta(n \log n)$ to…

$\Theta(n)$.

So…yeah….

We can do better!

In exchange for using auxiliary arrays (i.e. a not in-place sort).

Probably not better today. But maybe eventually…

Parallel Quick Sort

The bottleneck of the code isn't the number of recursive calls

It's the amount of time we spend doing the partitions.

Can we partition in parallel?

What is a partition?

It's moving all the elements smaller than the pivot into one subarray

And all the other elements into the other

Better Parallel Quick Sort

Sounds like a pack! (or two)

Step 1: choose pivot

Step 2: parallel pack elements smaller than pivot into the auxiliary array

Step 3: parallel pack elements greater than pivot into the auxiliary array

Step 4: Recurse! (in parallel)

Better Parallel Quick Sort

What is (a recurrence for)

The work?

The span?

Better Parallel Quick Sort

What is (a recurrence for)

The work: $\mathrm{T}_1(\mathrm{n}) = \begin{cases} 2T_1\left(\frac{n}{2}\right) + c_1 \cdot n & \text{if } n \geq \text{cutoff} \\ c_2 & \text{if } n < \text{cutoff} \end{cases}$

The span: $T_\infty(n) = \begin{cases} T_\infty\left(\frac{n}{2}\right) + c_1 \cdot \log n & \text{if } n \geq \text{cutoff} \\ c_2 & \text{if } n < \text{cutoff} \end{cases}$

Better Parallel Quick Sort

What is (a recurrence for)

The work: $\Theta(n \log n)$

The span: $\Theta(\log^2 n)$

Parallel Merge Sort

How do we merge?

Find median of one array. (in $O(1)$ time)

Binary search in the other to find where it would fit.

Merge the two left subarrays and the two right subarrays

- In parallel!

Only need one auxiliary array

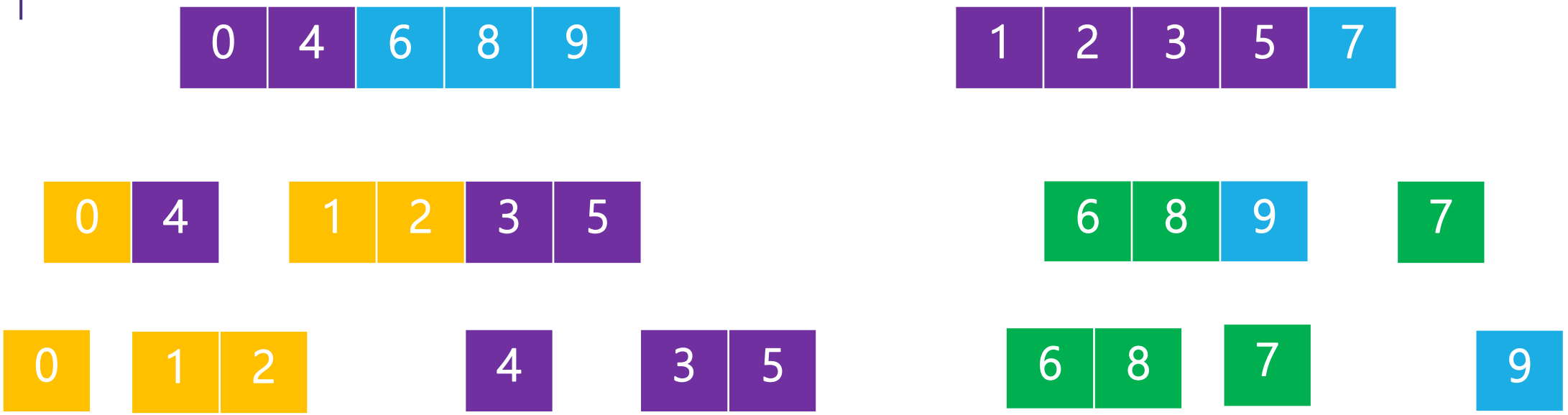
Each recursive call knows which range of the output array it's responsible for.

Key for the analysis: find the median in the bigger array, and binary search in the smaller array.

Parallel Merge

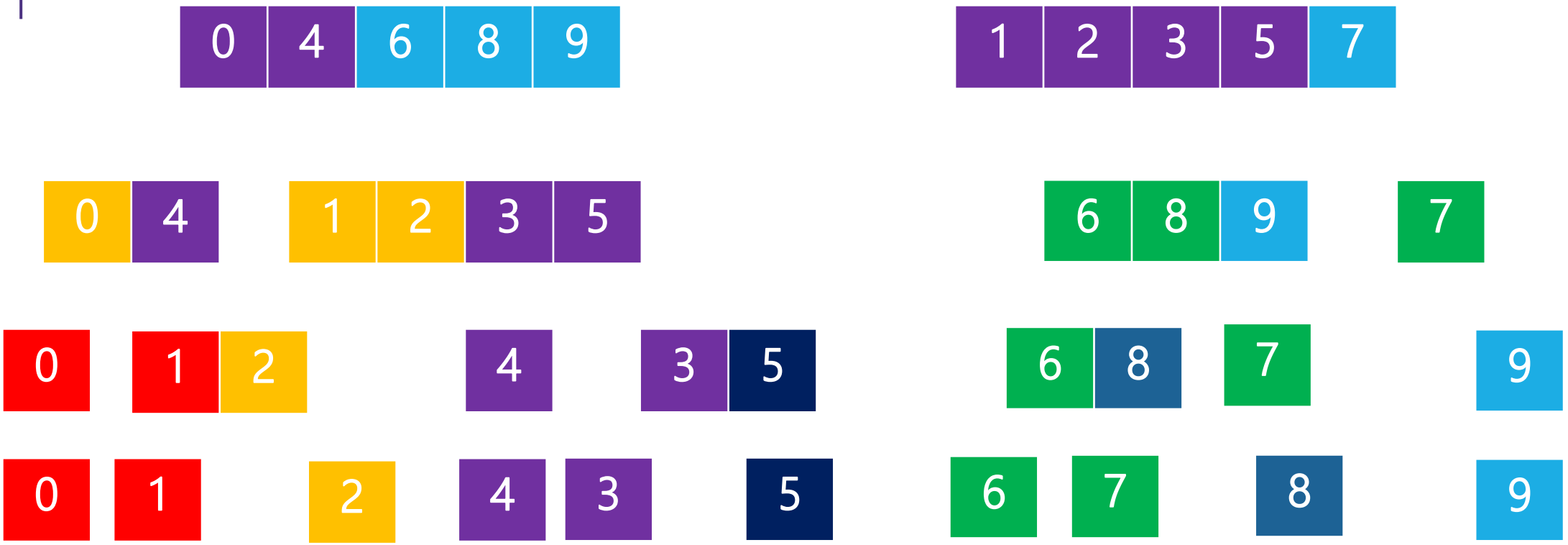
Find median of larger subarray (left chosen arbitrarily for tie)
Binary search to find where 6 fits in other array
Parallel recursive calls to merge

Parallel Merge



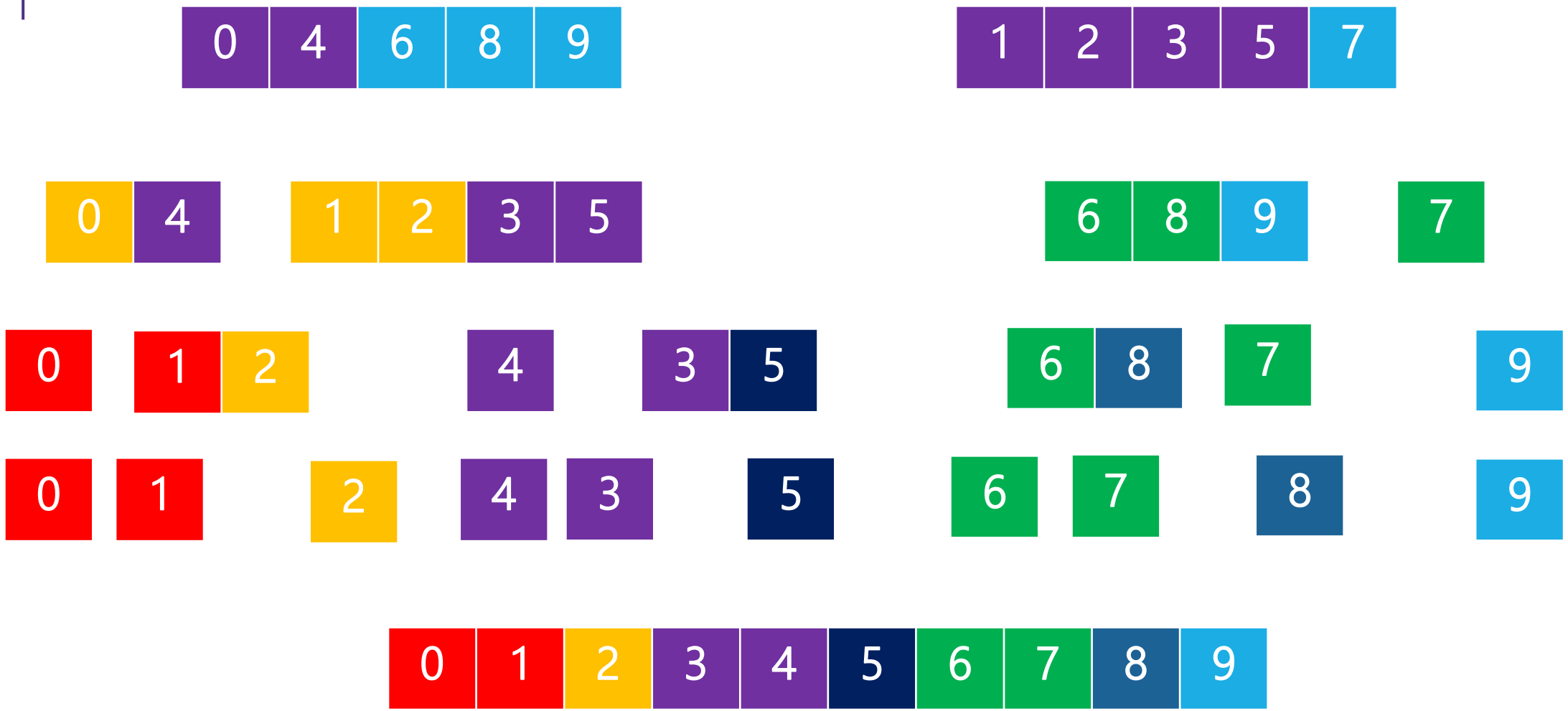
Find median of larger subarray (left chosen arbitrarily for tie)
Binary search to find where 6 fits in other array
Parallel recursive calls to merge

Parallel Merge



Find median of larger subarray (left chosen arbitrarily for tie)
Binary search to find where 6 fits in other array
Parallel recursive calls to merge

Parallel Merge



Parallel Merge Sort

Let's just analyze the merge:

What's the worst case?

One subarray has ¾ of the elements, the other has ¼ .

This is why we start with the median of the larger array.

Work: $\mathrm{T}_1(\mathrm{n}) = \begin{cases} T_1\left(\frac{3n}{4}\right) + T_1\left(\frac{n}{4}\right) + c \cdot \log n & \text{if } n \geq \text{cutoff} \\ c_2 & \text{if } n < \text{cutoff} \end{cases}$

Span: $T_\infty(n) = \begin{cases} T_\infty\left(\frac{3n}{4}\right) + c \cdot \log n & \text{if } n \geq \text{cutoff} \\ c_2 & \text{if } n < \text{cutoff} \end{cases}$

Parallel Merge Sort

Let's just analyze the merge:

What's the worst case?

One subarray has ¾ of the elements, the other has ¼ .

This is why we start with the median of the larger array.

Work: $\mathrm{T}_1(\mathrm{n}) = O(n)$

Span: $T_\infty(n) = O(\log^2 n)$

Parallel Merge Sort

Now the full mergesort algorithm:

Work: $\mathrm{T}_1(\mathrm{n}) = \begin{cases} 2T_1\left(\frac{n}{2}\right) + c \cdot n & \text{if } n \geq \text{cutoff} \\ c_2 & \text{if } n < \text{cutoff} \end{cases}$

Span: $T_\infty(n) = \begin{cases} T_\infty\left(\frac{n}{2}\right) + c \cdot \log^2 n & \text{if } n \geq \text{cutoff} \\ c_2 & \text{if } n < \text{cutoff} \end{cases}$

Parallel Merge Sort

Now the full mergesort algorithm:

Work: $\mathrm{T}_1(\mathrm{n}) = \Theta(n \log n)$

Span: $T_\infty(n) = \Theta(\log^3 n)$