Lecture 20: Parallel Prefix and Pack

CSE 332: Data Structures & Parallelism

Yafqa Khan

Summer 2025

Announcements

- EX09 due today
- EX10 due Friday
- Exam 2 information posted here:
  - https://courses.cs.washington.edu/courses/cse332/25su/exams/final.html
  - **Note: it will be hard to accommodate makeups; only four days to grade**
  - If you can’t make proposed makeup dates (e.g., sickness/emergency), some options:
  - Option 1: Exam 1 is worth 40% instead of 20% of overall grade
  - Option 2: Take the final exam in the next CSE 332 offering

Today

- Parallelism vs Concurrency
- Java Libraries for Parallelism
    - Java Thread Library
    - Java ForkJoin Library
- Simple Parallel Patterns (Algorithms)
    - Reductions
    - Maps
- Analyzing Parallel Algorithms
    - Work and Span
    - Amdahl's Law
- Fancier Parallel Patterns (Algorithms)
    - Prefix
    - Pack

Today

- Parallelism vs Concurrency
- Java Libraries for Parallelism
  - Java Thread Library
  - Java ForkJoin Library
- Simple Parallel Patterns (Algorithms)
  - Reductions
  - Maps
- Analyzing Parallel Algorithms
  - Work and Span
  - Amdahl's Law
- Fancier Parallel Patterns (Algorithms)
  - Prefix
  - Pack

The prefix-sum problem

Given **int[] input**, produce **int[] output** where:

output[i] = **input[0]+input[1]+…+input[i]**

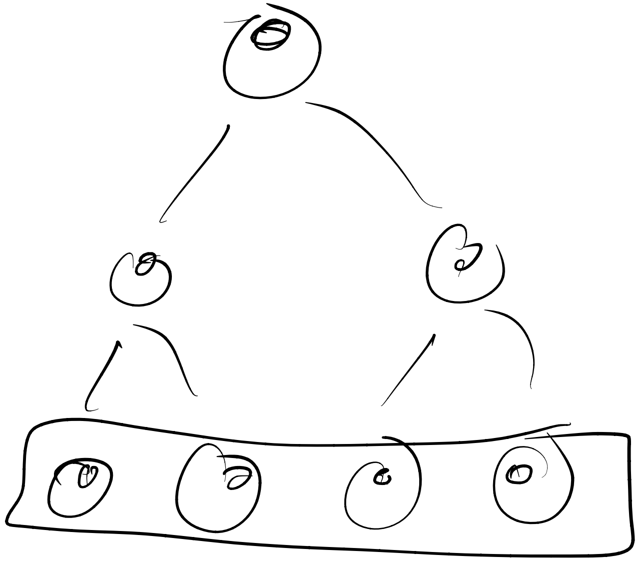
input	6	4	16	10	16	14	2	8
output	6	10	26	36	52	66	68	76

Sequential can be a CSE142 exam problem:

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

Parallel prefix-sum

- The parallel-prefix algorithm does two passes
  - Each pass has $O(n)$ work and $O(\texttt{log } n)$ span
  - So in total there is $O(n)$ work and $O(\texttt{log } n)$ span
  - So like with array summing, parallelism is $n/\texttt{log } n$
    - An exponential speedup



- First pass builds the tree bottom-up: the “up” pass

- Second pass traverses the tree top-down: the “down” pass

Local bragging

Historical note:

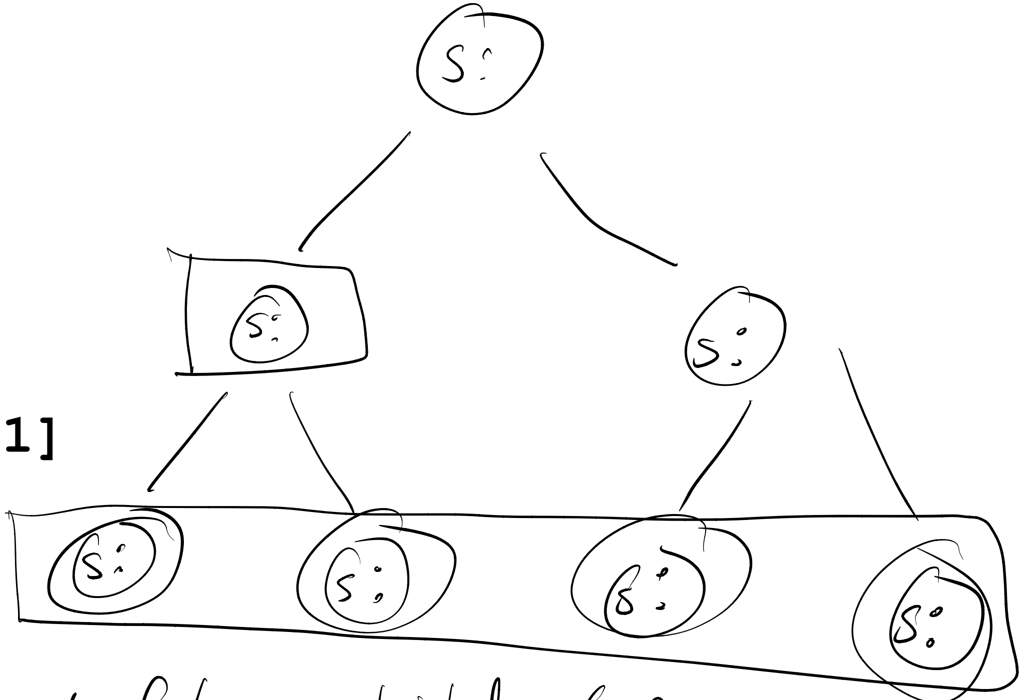
- Original algorithm due to R. Ladner and M. Fischer at UW in 1977
- Richard Ladner joined the UW faculty in 1971 and hasn't left

1968?

recent

The algorithm, part 1

1. Propagate 'sum' up: Build a binary tree where
   - Root has sum of **`input[0]..input[n-1]`**
   - Each node has sum of **`input[lo]..input[hi-1]`**
     - Build up from leaves
       - **`parent.sum = left.sum + right.sum`**
   - A leaf's sum is just it's value
     - **`leaves[i].sum = input[i]`**



- build left child (lo, mid)
- buid right child (mid, hi)

making the root node

This is an easy fork-join computation: same as sum algorithm of array but this time store answers in tree as we move up

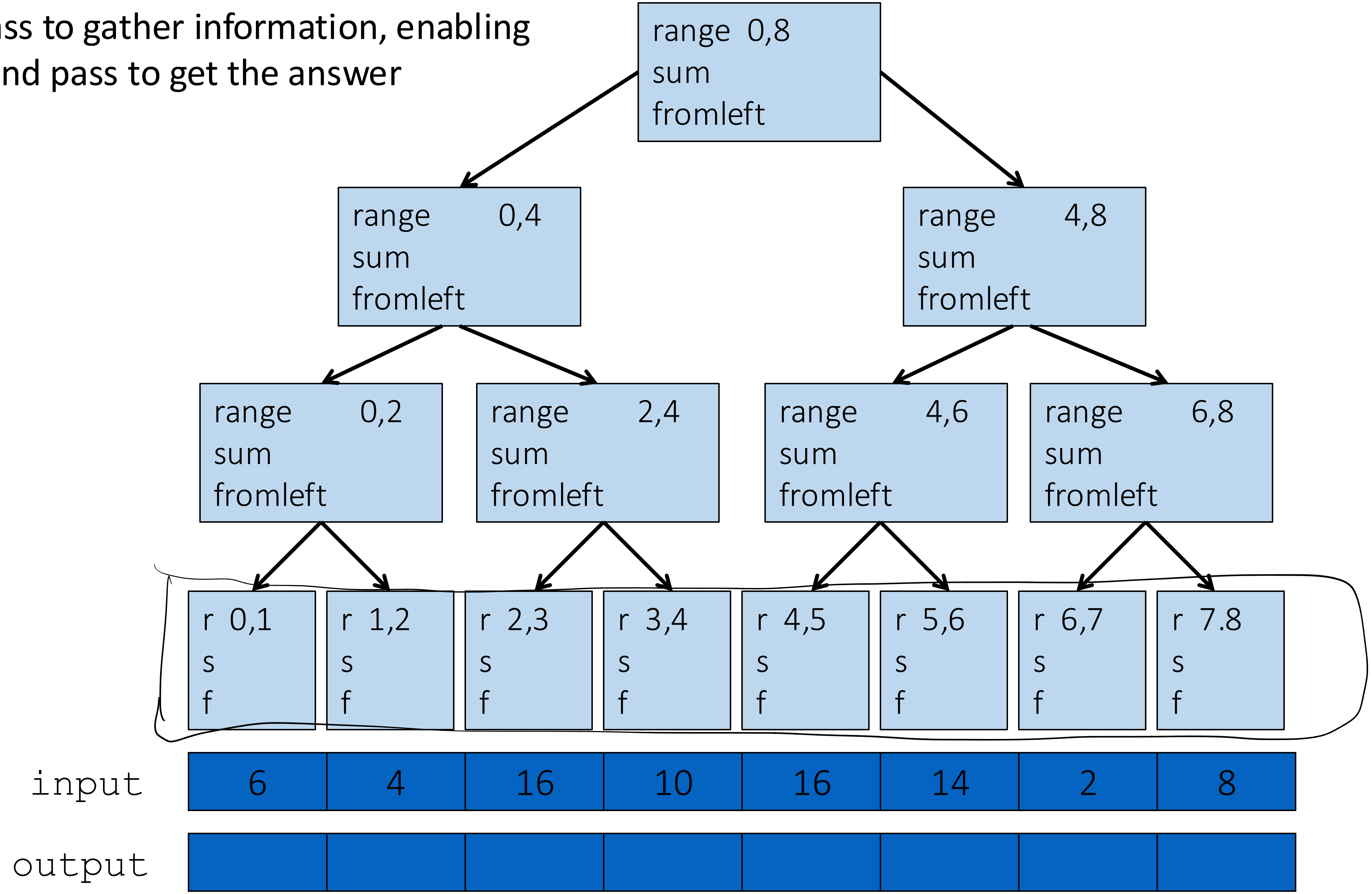
- set sum to be left.sum + right.sum

The (completely non-obvious) idea:
Do an initial pass to gather information, enabling us to do a second pass to get the answer

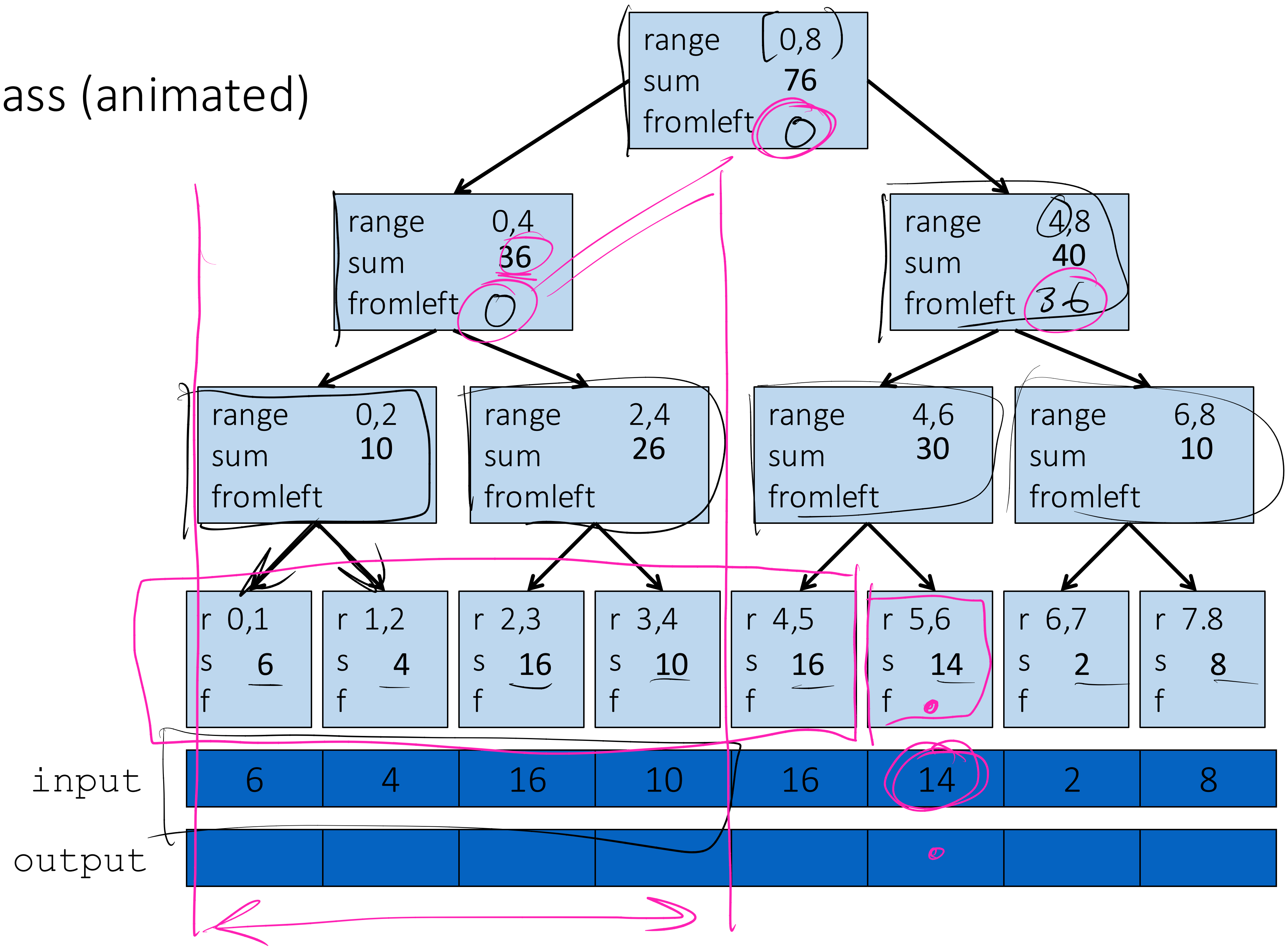
range 0,8
sum
fromleft

input								
	6	4	16	10	16	14	2	8

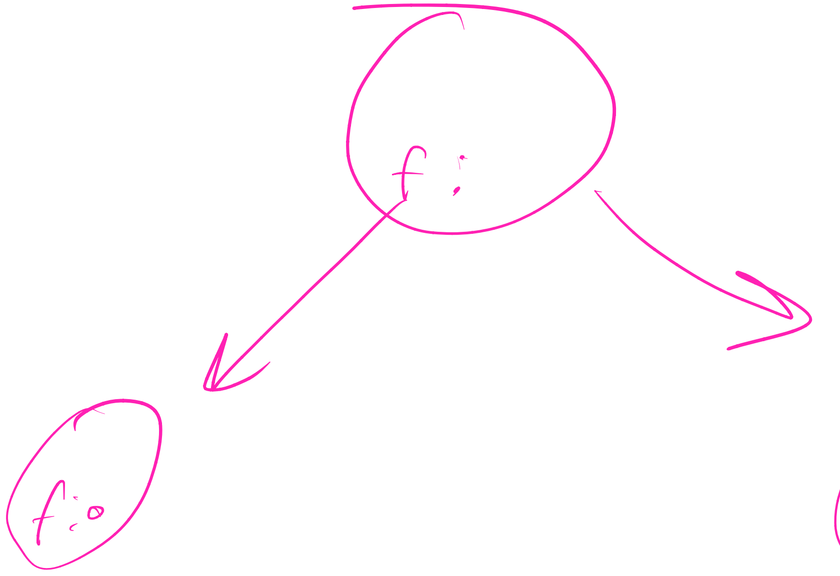
The (completely non-obvious) idea:
Do an initial pass to gather information, enabling us to do a second pass to get the answer



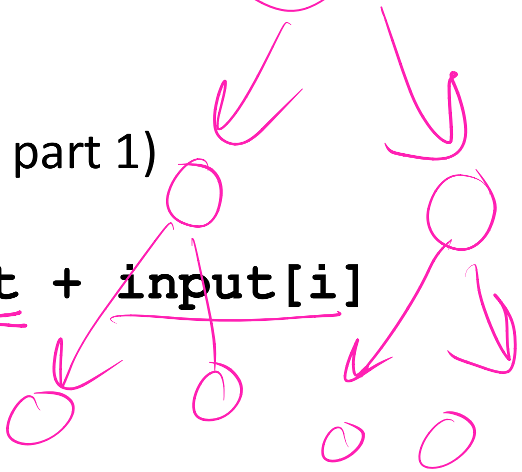
First pass (animated)



The algorithm, part 2



2. Propagate ‘fromleft’ down:
   - Root given a **`fromLeft`** of **0**
   - Node takes its **`fromLeft`** value and
     - Passes its left child the same **`fromLeft`**
       - **`node.left.fromLeft = node.fromleft`** 1
     - Passes its right child its **`fromLeft`** plus its left child’s **`sum`** (as stored in part 1)
       - **`node.right.fromLeft = node.fromleft + node.left.sum`** 2
   - At the leaf for array position **`i`**, **`output[i] = leaves[i].fromLeft + input[i]`**

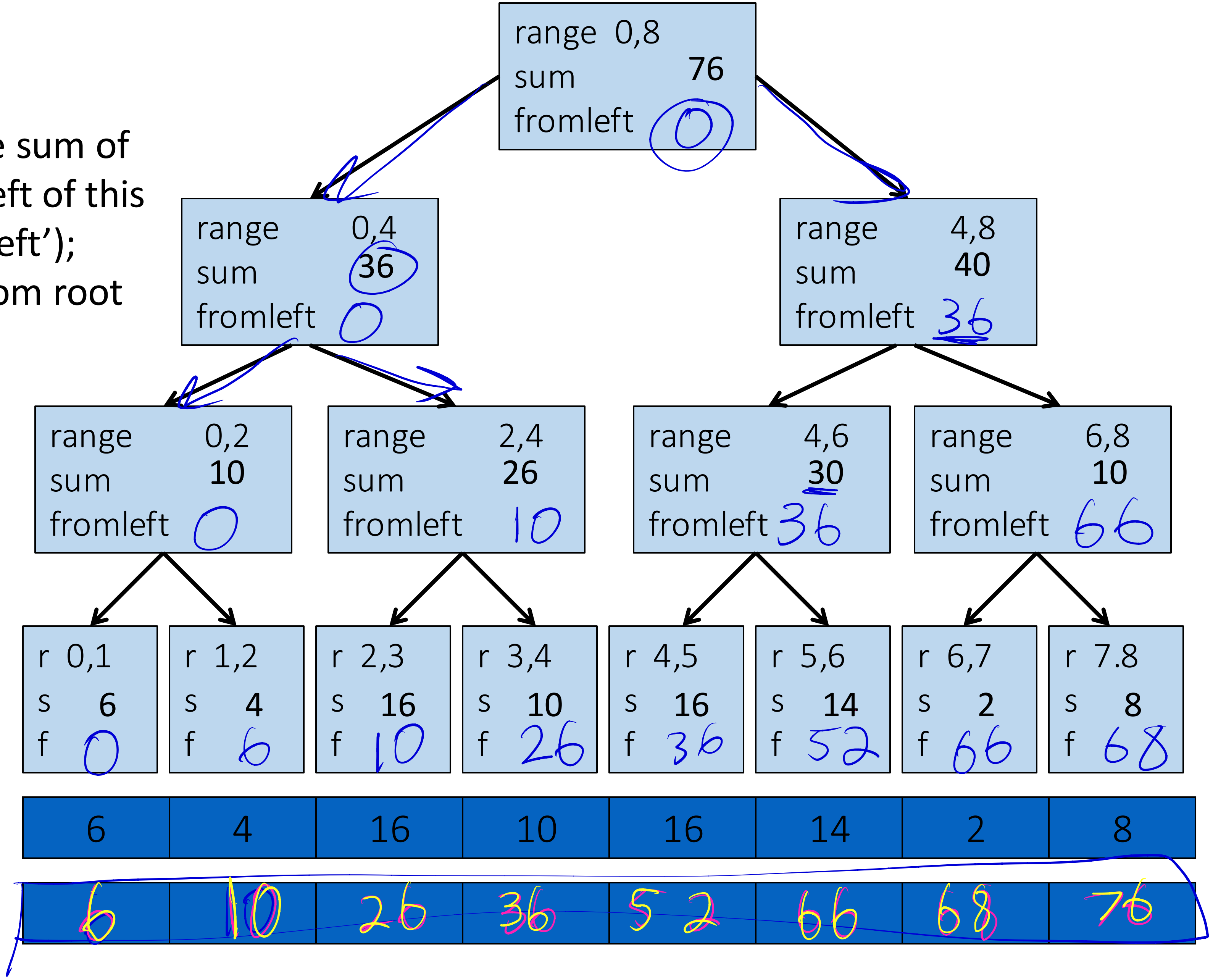


This is also an easy fork-join computation: traverse the tree built in step 1 and fill in the fromLeft field using saved information

- Invariant: **`fromLeft`** is sum of elements left of the node’s range

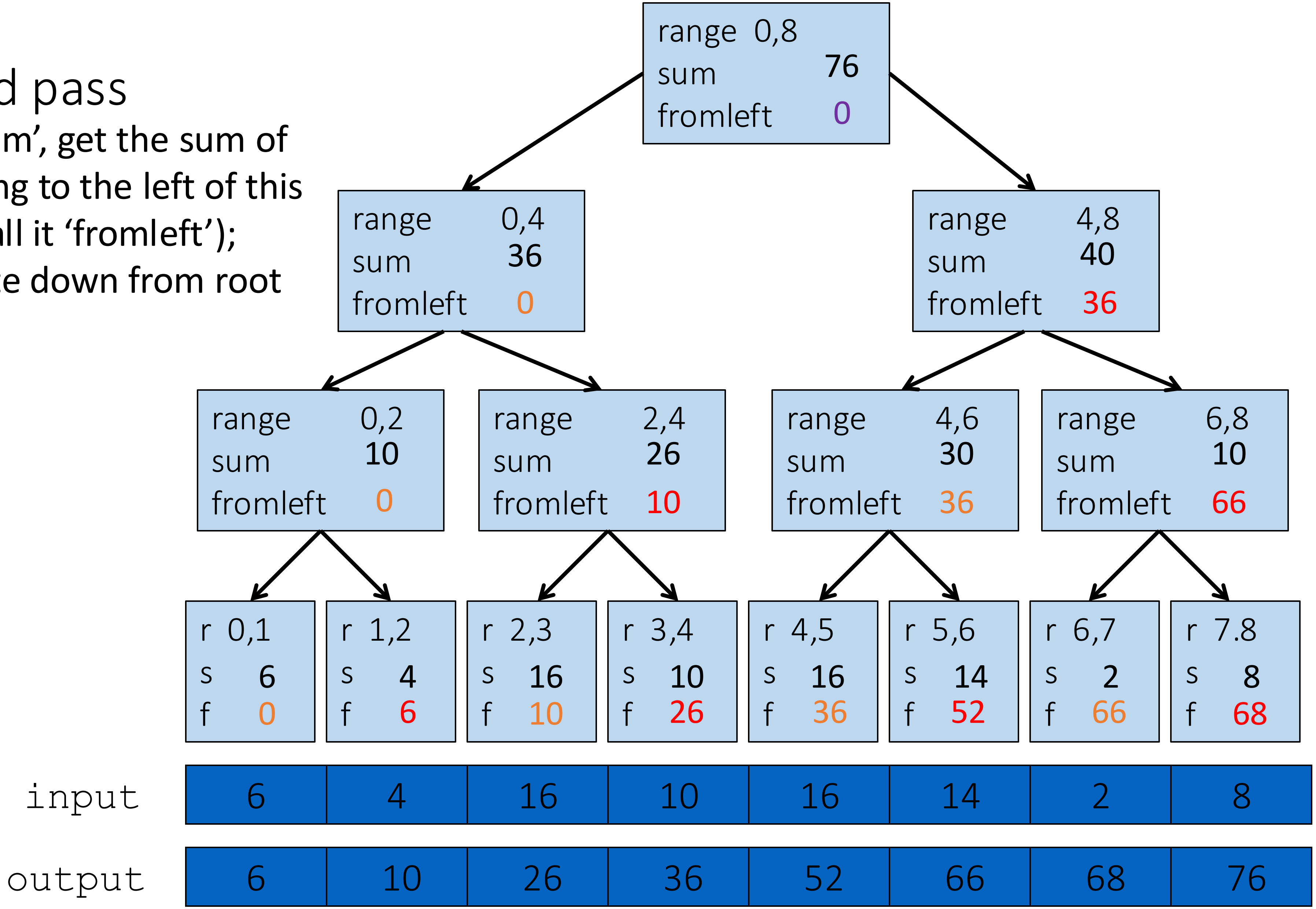
Second pass

Using 'sum', get the sum of everything to the left of this range (call it 'fromleft'); propagate down from root



Second pass

Using 'sum', get the sum of everything to the left of this range (call it 'fromleft'); propagate down from root



$T_\infty(n) = T_\infty(n/2) + c$

Analysis of Algorithm

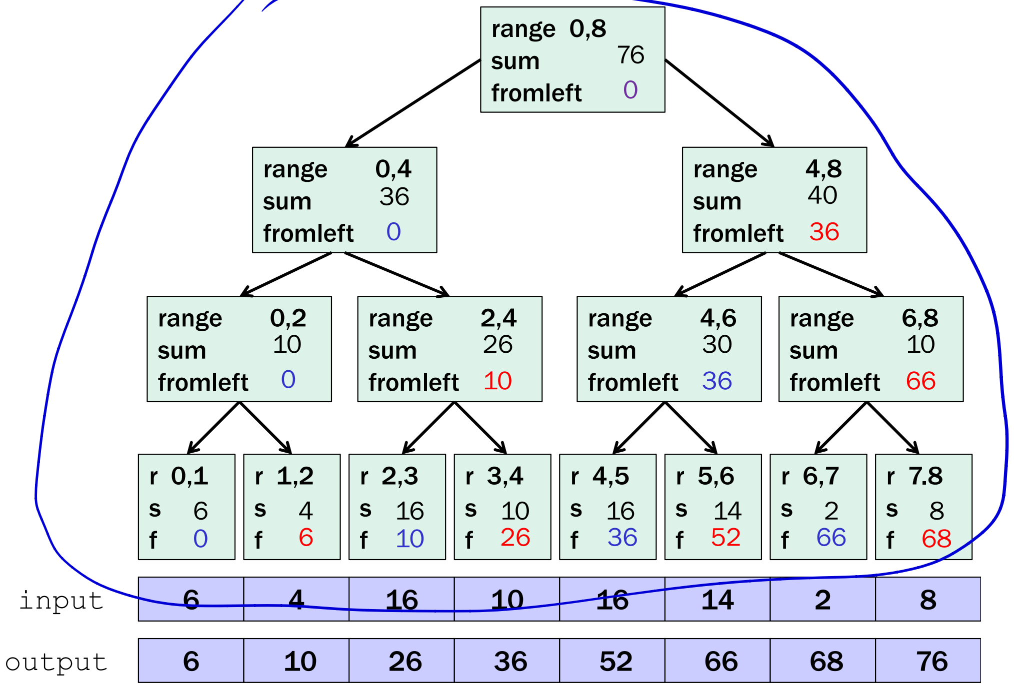
$T_\infty(n) \in O(\log(n))$

Original boring 142 algorithm: *O(n)*

Analysis of our fancy prefix sum algorithm:

Analysis of first step: $O(n)$ work

$O(\log n)$ span



Analysis of second step:

$T_\infty(n) = T_\infty(n/2) + c \Rightarrow O(\log n)$ span

$O(n)$ work

Total for algorithm:

$O(n)$ work, $O(\log n)$ span

Analysis of Algorithm

Original boring 142 algorithm: *O(n)*

Analysis of our fancy prefix sum algorithm:

Analysis of first step:

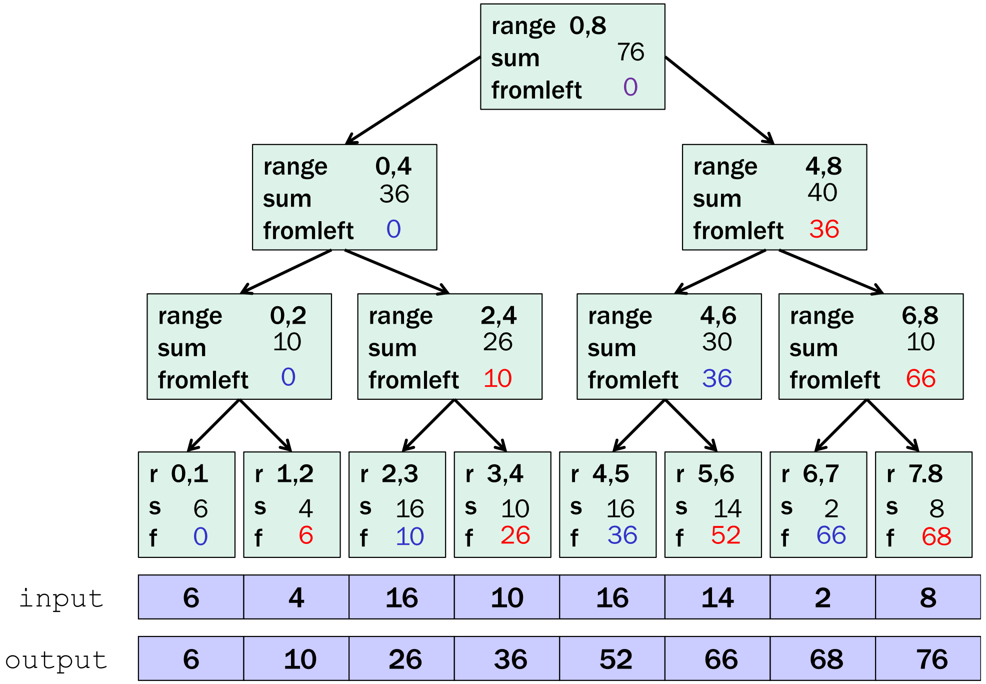
$O(n)$ **work**, $O(\log n)$ **span**

Analysis of second step:

$O(n)$ **work**, $O(\log n)$ **span**

Total for algorithm:

$O(n)$ **work**, $O(\log n)$ **span**



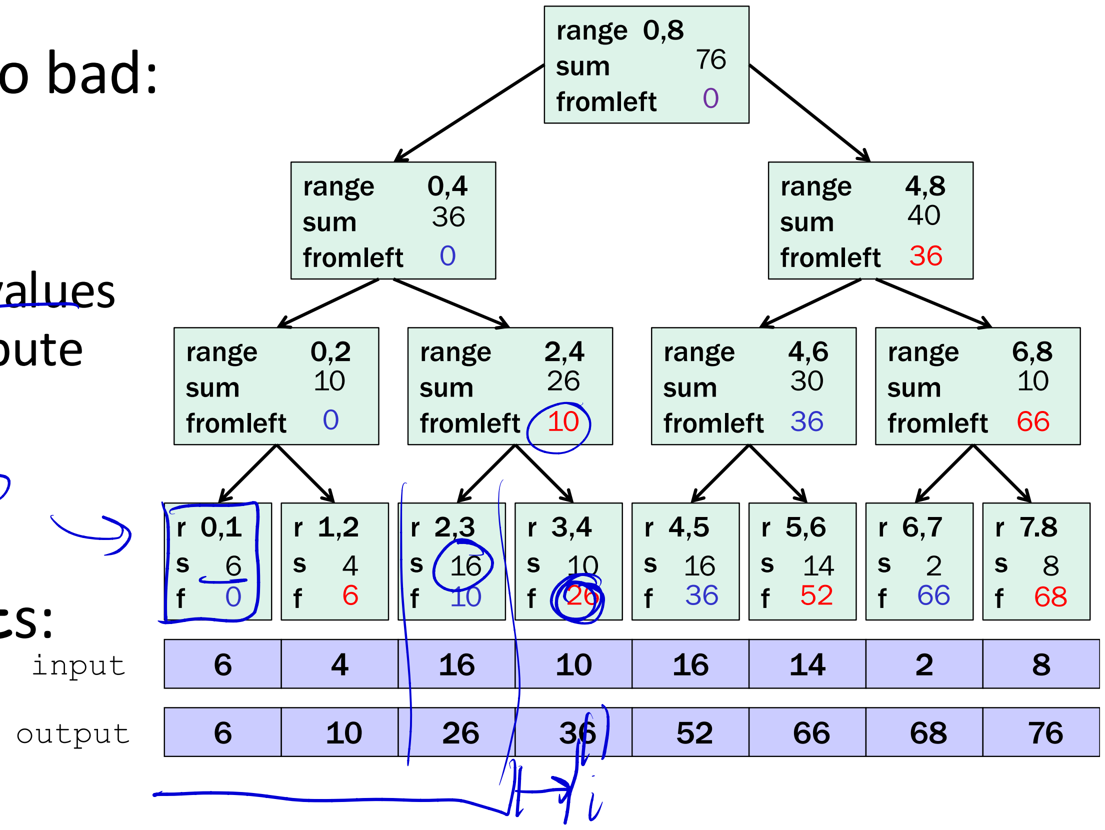
Sequential cut-off

Optimizing: Adding a sequential cut-off isn't too bad:

- **Step One**: Propagating Up the **`sums`**:
  - Have a leaf node just hold the sum of a range of values instead of just one array value (Sequentially compute sum for that range)
  - The tree itself will be shallower

0, 100

- **Step Two**: Propagating Down the **`fromLefts`**:
  - At leaf, compute prefix sum over its [lo,hi):



On the topic of optimization, do we need to actually have a tree?

Parallel prefix, generalized

Just as sum-array was the simplest example of a common pattern, prefix-sum illustrates a pattern that arises in many, many problems

- Minimum, maximum of all elements **to the left of `i`**
- Is there an element **to the left of `i`** satisfying some property?

- Count of elements **to the left of `i`** satisfying some property
  - This last one is perfect for an efficient parallel pack…
  - Perfect for building on top of the "parallel prefix trick"

Pack (i.e., "Filter")

Given an array **`input`**, produce an array **`output`** containing only elements such that **`f(element)`** is **`true`**

Example: **`input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]`**

`f: "is element > 10"`

`output [17, 11, 13, 19, 24]`

Parallelizable?

- Determining *whether* an element belongs in the output is easy
- But determining *where* an element belongs in the output is hard; seems to depend on previous results….

In this example, Filter = element > 10

Solution! Parallel Pack = parallel map + parallel prefix + parallel map

1. **Parallel map** to compute a bit-vector for true elements:

```
input   [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
bits    [ 1, 0, 0, 0,  1, 0,  1,  1, 0,  1]
```

2. **Parallel-prefix** sum ***on the bit-vector***:

```
bitsum  [ 1, 1, 1, 1,  2, 2,  3,  4, 4,  5]
```

3. **Parallel map** to produce the output:

```
output  [17, 11, 13, 19, 24]
```

each step is O(n) work O(log n) span

```
output = new array of size bitsum[n-1]
FORALL(i=0; i < input.length; i++){
  if(bits[i]==1)
    output[bitsum[i]-1] = input[i];
}
```

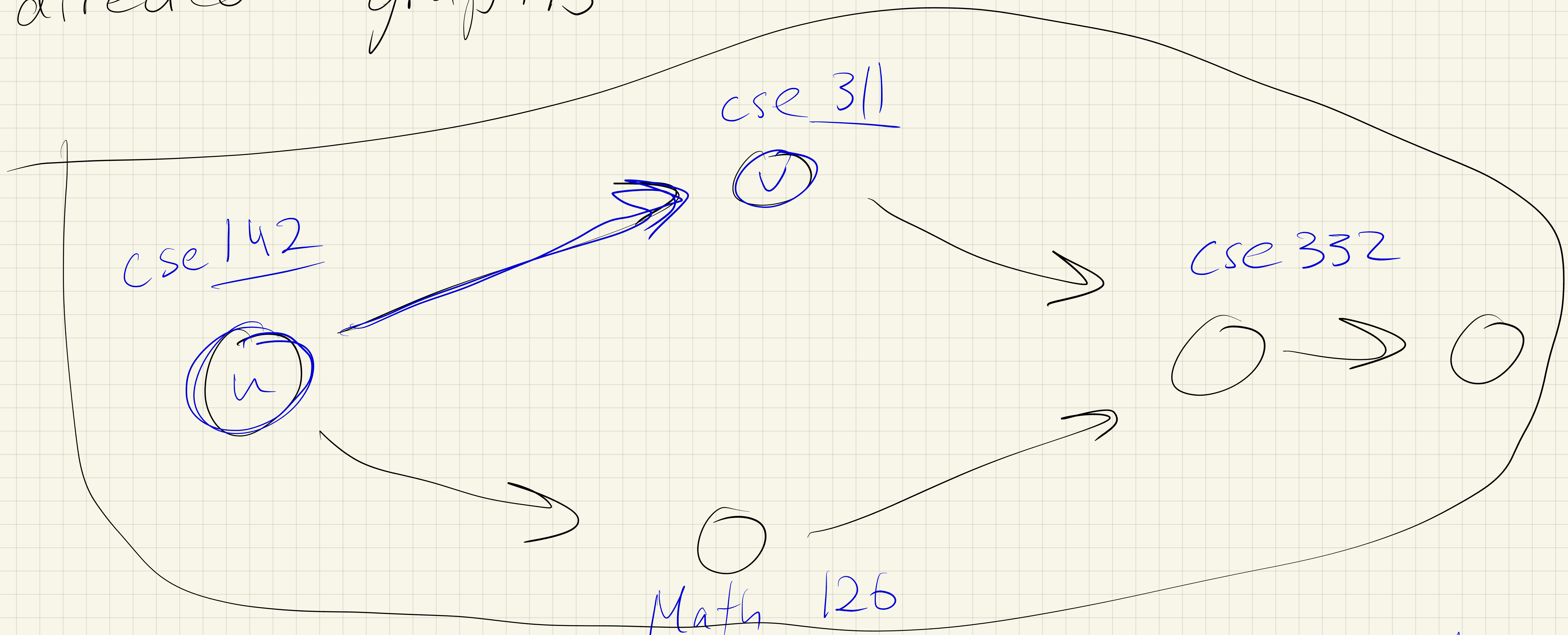
Pack comments

- First two steps can be combined into one pass
  - Just using a different base case for the prefix sum
  - No effect on asymptotic complexity
- Can also combine third step into the down pass of the prefix sum
  - Again no effect on asymptotic complexity
- Analysis: $O(n)$ work, $O(\texttt{log}\ n)$ span
  - 2 or 3 passes, but 3 is a constant ☺

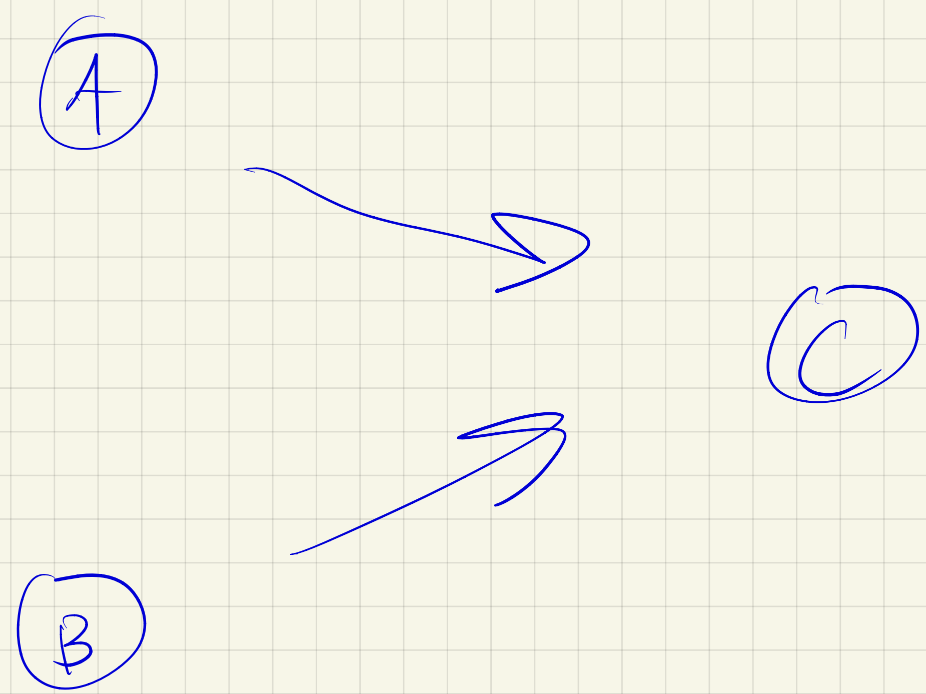
Any Questions?

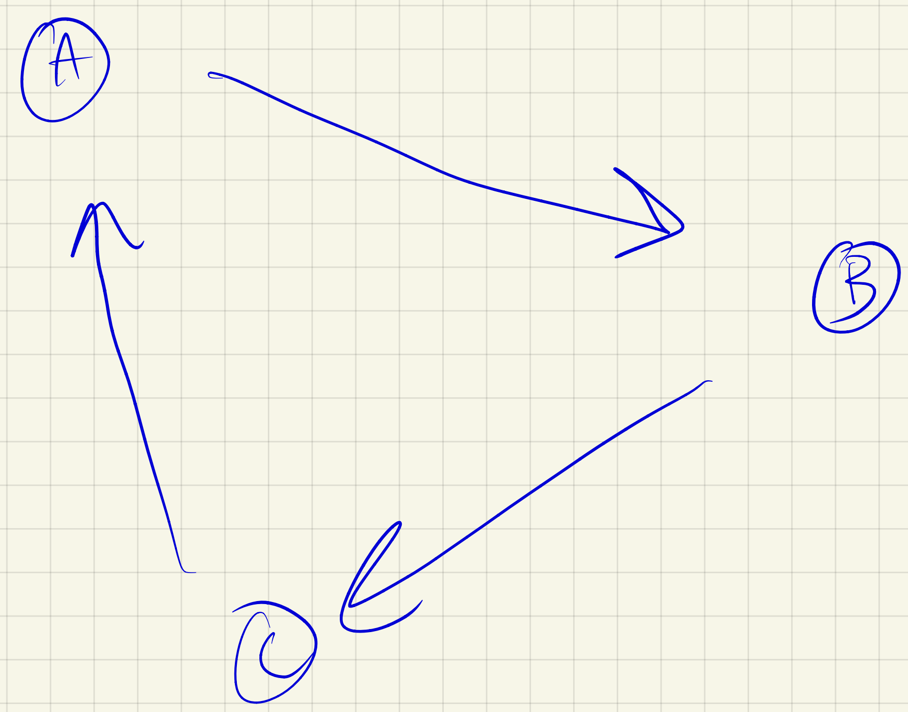
Graphs: topological ordering

- directed graphs



topological ordering: order of nodes s.t. if $(u, v) \in E$, then u comes before v.





A, B, C

B, A, C

$A \to B \to C \to A$

- Thm: a graph has a topological ordering iff the graph is a DAG.

- How to find a topo ordering in a DAG?

alg: 1) initialize indegrees

2) put all nodes u
indegree 0 into queue

3) while (!q.isEmpty) {
- remove from queue
- for each neighbor, v:
  - decrease v.indegree
  - if v.indegre == 0:
    - add v to queue