# CSE 332: Data Structures & Parallelism

# Lecture 19: Parallel Prefix & Pack

Ruth Anderson

Winter 2025

# *Outline*

Done:

- Simple ways to use parallelism for counting, summing, finding
- Analysis of running time and implications of Amdahl's Law

Now: Clever ways to parallelize more than is intuitively possible

- Parallel prefix:
  - This "key trick" typically underlies surprising parallelization
  - Enables other things like packs (aka filters)

# *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;
}
```

Does not seem parallelizable

- Work: $O(n)$, Span: $O(n)$
- *This algorithm* is sequential, but a *different algorithm* has Work: $O(n)$, Span: $O(\log n)$

# *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 a 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? 1973?

recent

# *Parallel Prefix: The Up Pass*

We build want to <u>build a binary tree</u> where

- Root has sum of the range [x,y)
- If a node has sum of [lo,hi) and hi>lo,
  - Left child has sum of [lo,middle)
  - Right child has sum of [middle,hi)
  - A leaf has sum of [i,i+1), which is simply input[i]

It is critical that we actually <u>create the tree</u> as we will need it for the down pass

- We do not need an actual linked structure
- We could use an array as we did with heaps

Analysis of first step: Work = Span =

# *The algorithm, part 1*

Specifically…..

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; **`input[i]`**

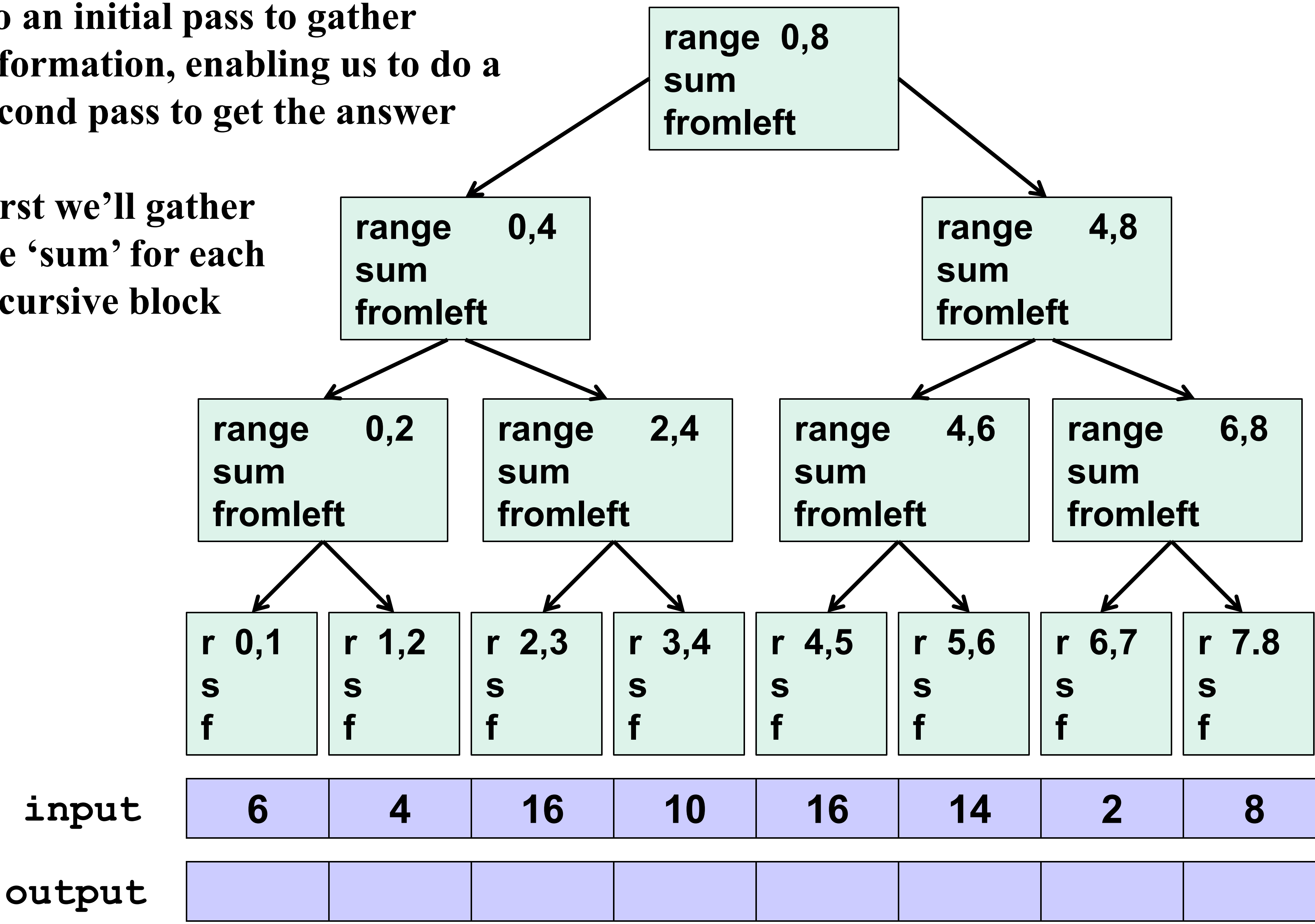
This is an easy fork-join computation: combine results by actually building a binary tree with all the sums of ranges

- Tree built bottom-up in parallel
- Could be more clever; ex. Use an array as tree representation like we did for heaps

Analysis of first step: *O*(*n*) **work**, *O*(**`log`** *n*) **span**

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

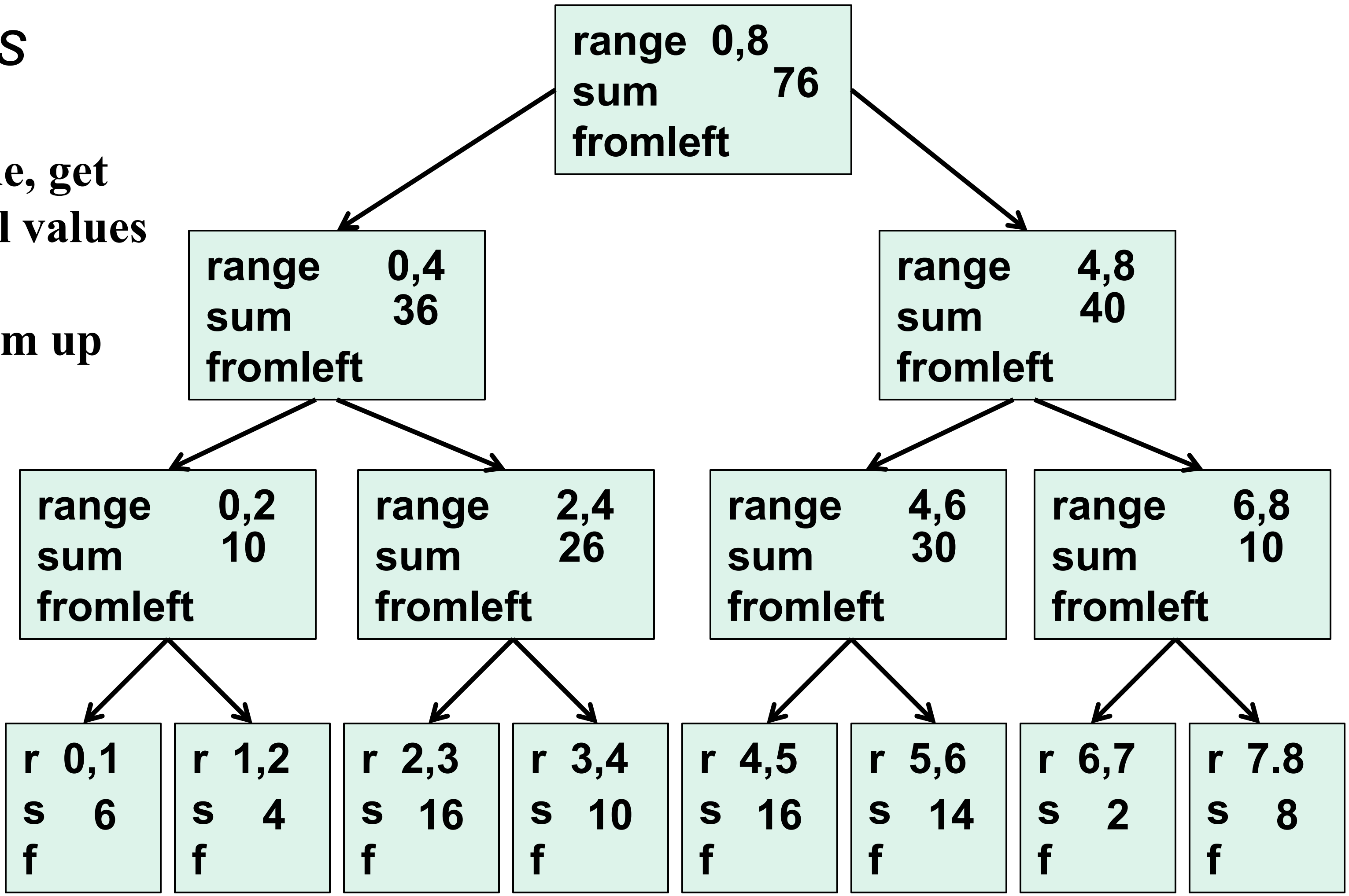
**First we'll gather the 'sum' for each recursive block**



# *First pass*

**For each node, get the sum of all values in its range; propagate sum up from leaves**

**Will work like parallel sum, but recording intermediate information**




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| `input` | 6 | 4 | 16 | 10 | 16 | 14 | 2 | 8 |
| `output` | | | | | | | | |

# *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`**
     - Passes its right child its **`fromLeft`** plus its left child’s **`sum`** (as stored in part 1)
   - At the leaf for array position **`i`**, **`output[i]=fromLeft+input[i]`**

This is an easy fork-join computation: traverse the tree built in step 1 and produce no result (the leaves assign to **`output`**)

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

Analysis of first step: $O(n)$ **work**, $O(\log n)$ **span**

Analysis of second step:

**Total for algorithm:**

# *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`**
     • Passes its right child its **`fromLeft`** plus its left child’s **`sum`** (as stored in part 1)
   – At the leaf for array position **`i`**, **`output[i]=fromLeft+input[i]`**

This is an easy fork-join computation: traverse the tree built in step 1 and produce no result (the leaves assign to **`output`**)
   – Invariant: **`fromLeft`** is sum of elements left of the node’s range

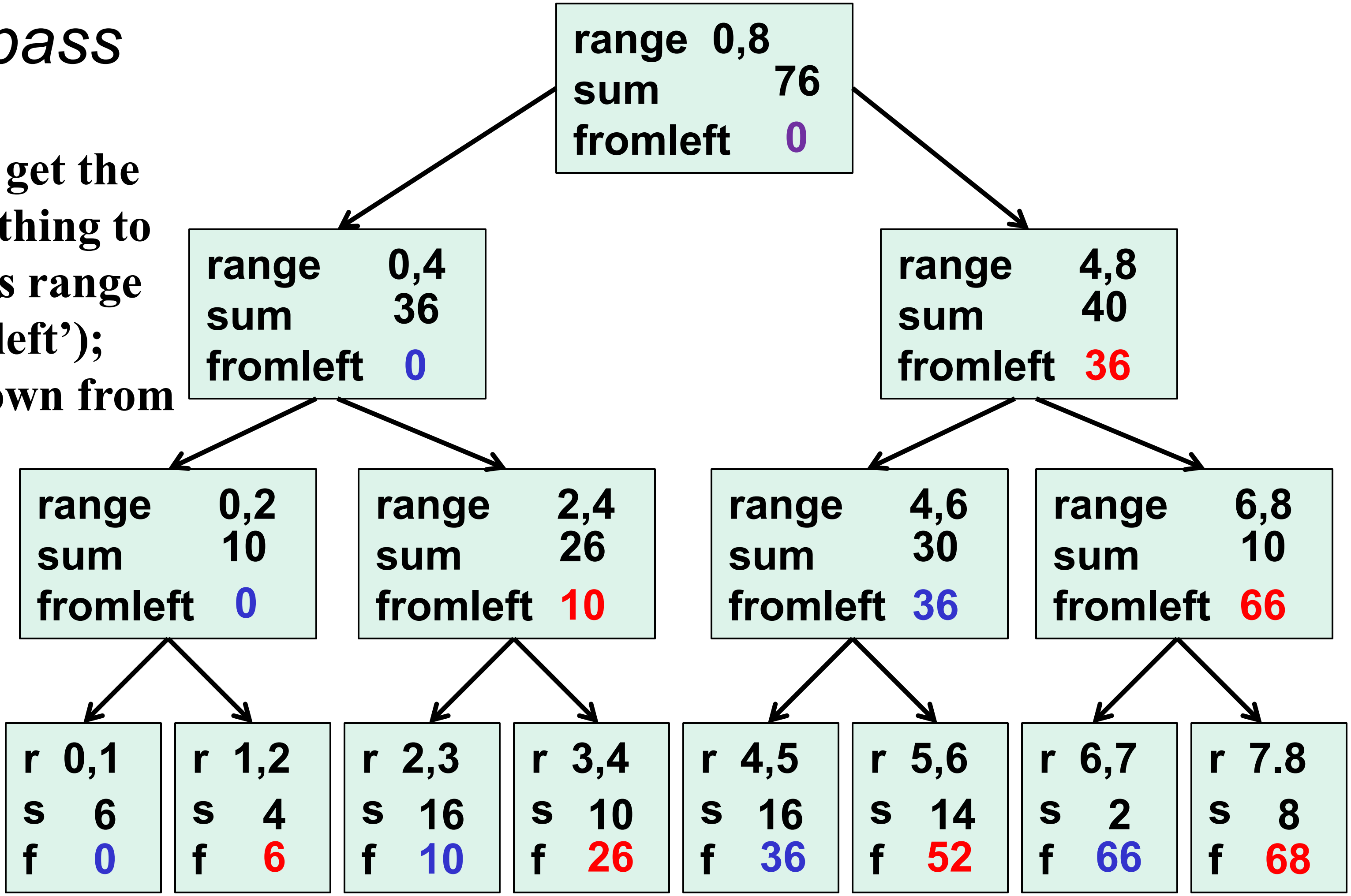
Analysis of first step: $O(n)$ **work**, $O(\texttt{log}\ n)$ **span**

Analysis of second step: $O(n)$ **work**, $O(\texttt{log}\ n)$ **span**

**Total for algorithm:** $O(n)$ **work**, $O(\texttt{log}\ n)$ **span**

# *Second pass*

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




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

# *Sequential cut-off*

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

```
output[lo] = fromLeft + input[lo];
for(i=lo+1; i < hi; i++)
  output[i] = output[i-1] + input[i]
```

# *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 (think "Filter")*

[Non-standard terminology]

Given an array **`input`**, produce an array **`output`** containing <u>only</u> 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 <u>*whether*</u> an element belongs in the output is easy
- But determining <u>*where*</u> an element belongs in the output is hard; seems to depend on previous results….

In this example, Filter = element > 10

# *Parallel Pack = (Soln)*
# *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]
```

```
output = new array of size bitsum[n-1]
FORALL(i=0; i < input.length; 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, $O(\mathtt{log}\ n)$ span
  - 2 or 3 passes, but 3 is a constant ☺
- Parallelized packs will help us parallelize quicksort. (see reading)