Parallel Programming

Task Graphs, Parallel Patterns, Parallel Algorithms

What we have seen so far

Optimizing sequential or parallel code by considering hardware design (L06)

Using Java threads to take advantage of multiple cores (L03-05)

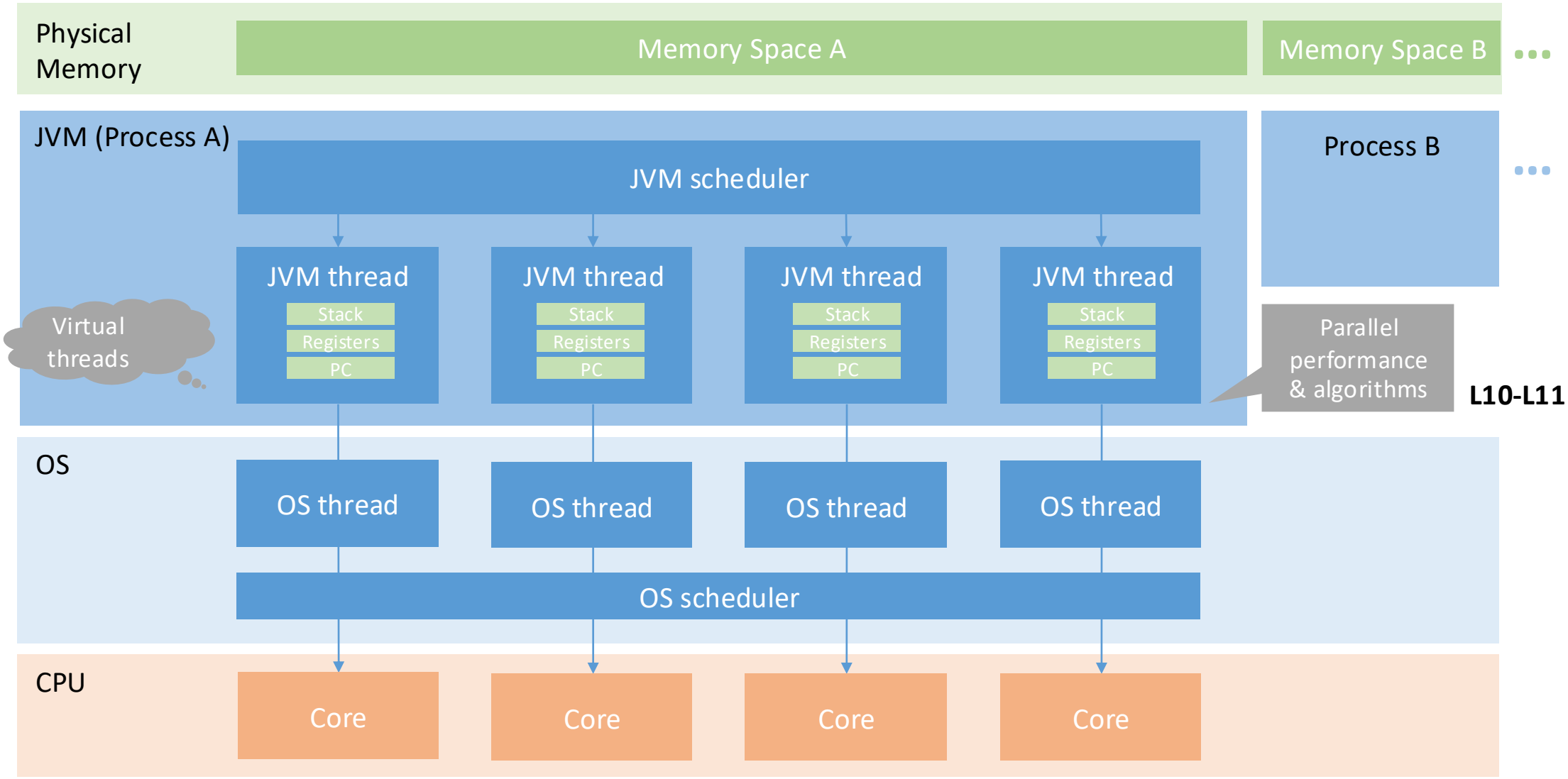
D&Q for work distribution and FJ for thread assignment (and executor for flat structures) (L08-09)

Scalability, metrics, and Amdahl's Law (L07)

Today: Connect performance metrics with D&Q and FJ

Next: Reduce sequential part: parallel patterns and algorithms

Big picture (part I)



Structure of next lectures

Task graphs

Performance measures

Parallel patterns

Parallel algorithms

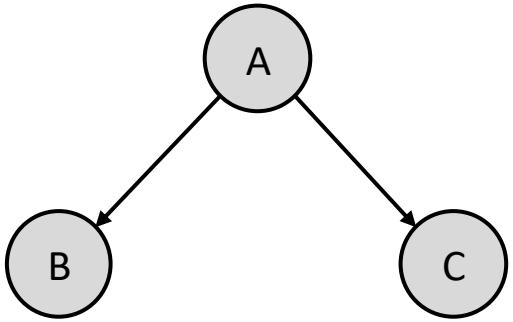
Task graphs and performance measures

Task Parallel Programming (Cilk-style)

Tasks:

- Execute code
- Spawn other tasks (= start in JT, submit in ExS, fork in FJ)
- Wait for results from other tasks (= join in JT and FJ, get in ExS)

A graph is formed based on spawning tasks



The edges mean:

- Task B was created by Task A
- Task C was created by Task A

Directed acyclic graph (DAG)

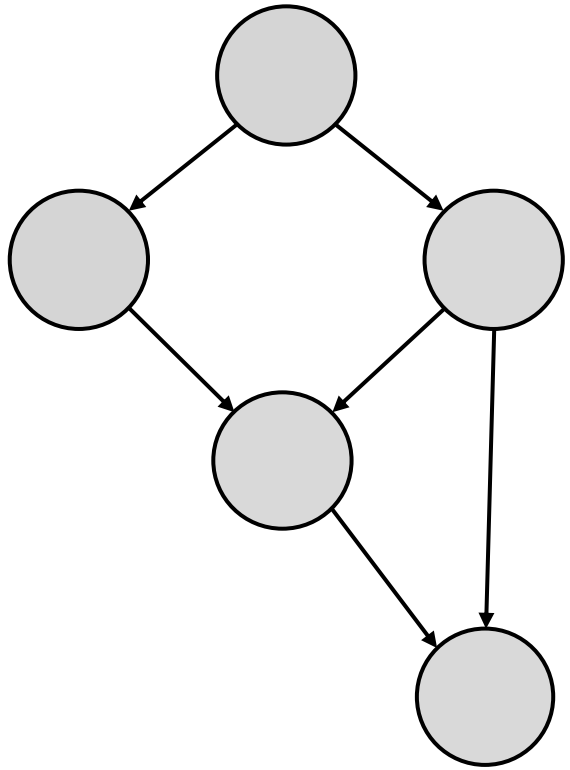
A program execution using **`fork`** and **`join`** can be seen as a DAG

Nodes: Units of work

- FJ: tasks; Cilk: finer-grained strands

Edges: Dependencies

- Source must finish before destination starts (logical completion, not immediate termination)

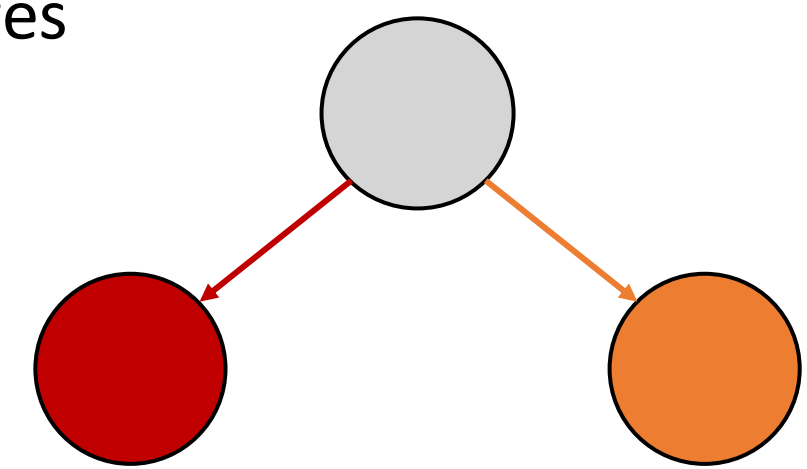


The DAG does not depict "who is running when”, but rather "who must finish before what"

Directed acyclic graph (DAG)

A `fork` ends a node and creates two outgoing edges

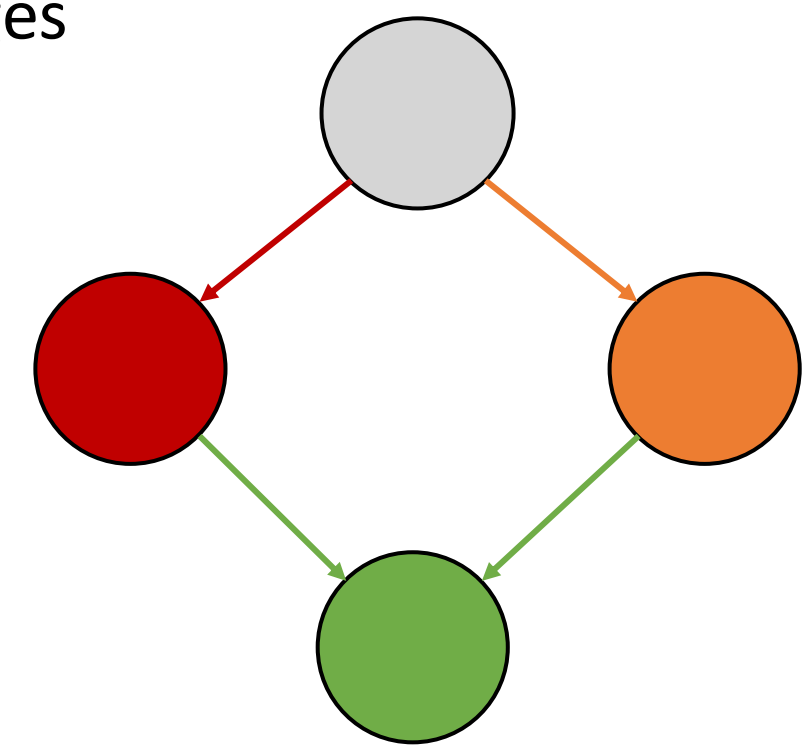
- One to a new forked task
- One to the continuation of the current task
- May execute in parallel if worker thread is available



Directed acyclic graph (DAG)

A `fork` ends a node and creates two outgoing edges

- One to a new forked task
- One to the continuation of the current task
- May execute in parallel if worker thread is available



A `join` creates a node with two incoming edges

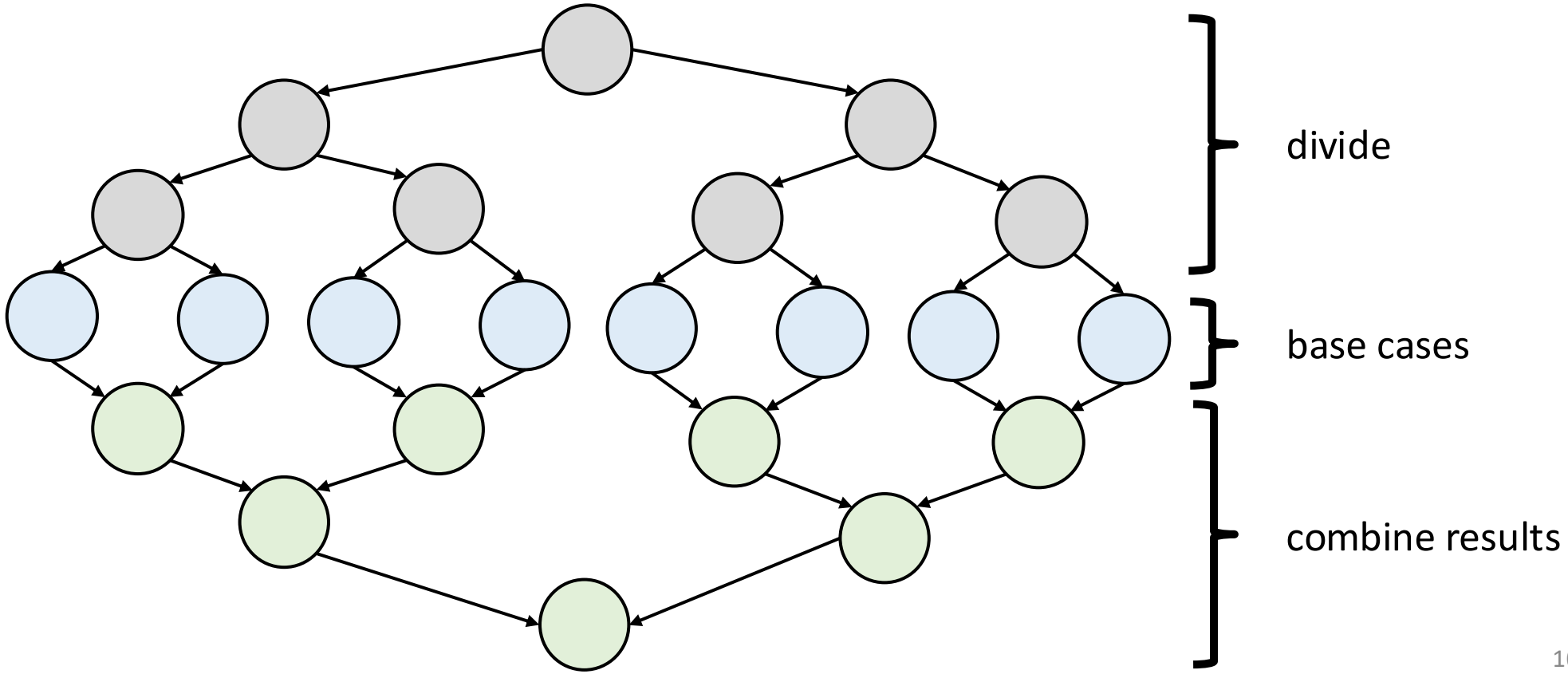
- One from the forked task that just finished
- One from the continuation of the original task (which was waiting for the forked task to complete)

The continuation does not execute until both branches leading into the join node are complete

Balanced tree D&C example

`fork-join` is highly flexible

A tree of recursive calls followed by an upside-down tree of result combination, used in many algorithms



Task parallelism discussion

- The task graph is **dynamic** - it unfolds as execution proceeds
- Tasks (represented by nodes) can execute in **parallel**
- But they don’t have to - tasks are assigned to worker threads dynamically as they become available
- Work-stealing allows idle threads to "steal" tasks from busier threads to balance the workload

fib() function

$$fib(n) = \begin{cases} n, & \text{if } n \leq 1 \\ fib(n-1) + fib(n-2), & \text{if } n > 1 \end{cases}$$

Sequential version

```
public class SequentialFib {
    public static int fib(int n) {
        if (n <= 1) return n;

        return fib(n - 1) + fib(n - 2);
    }
}
```

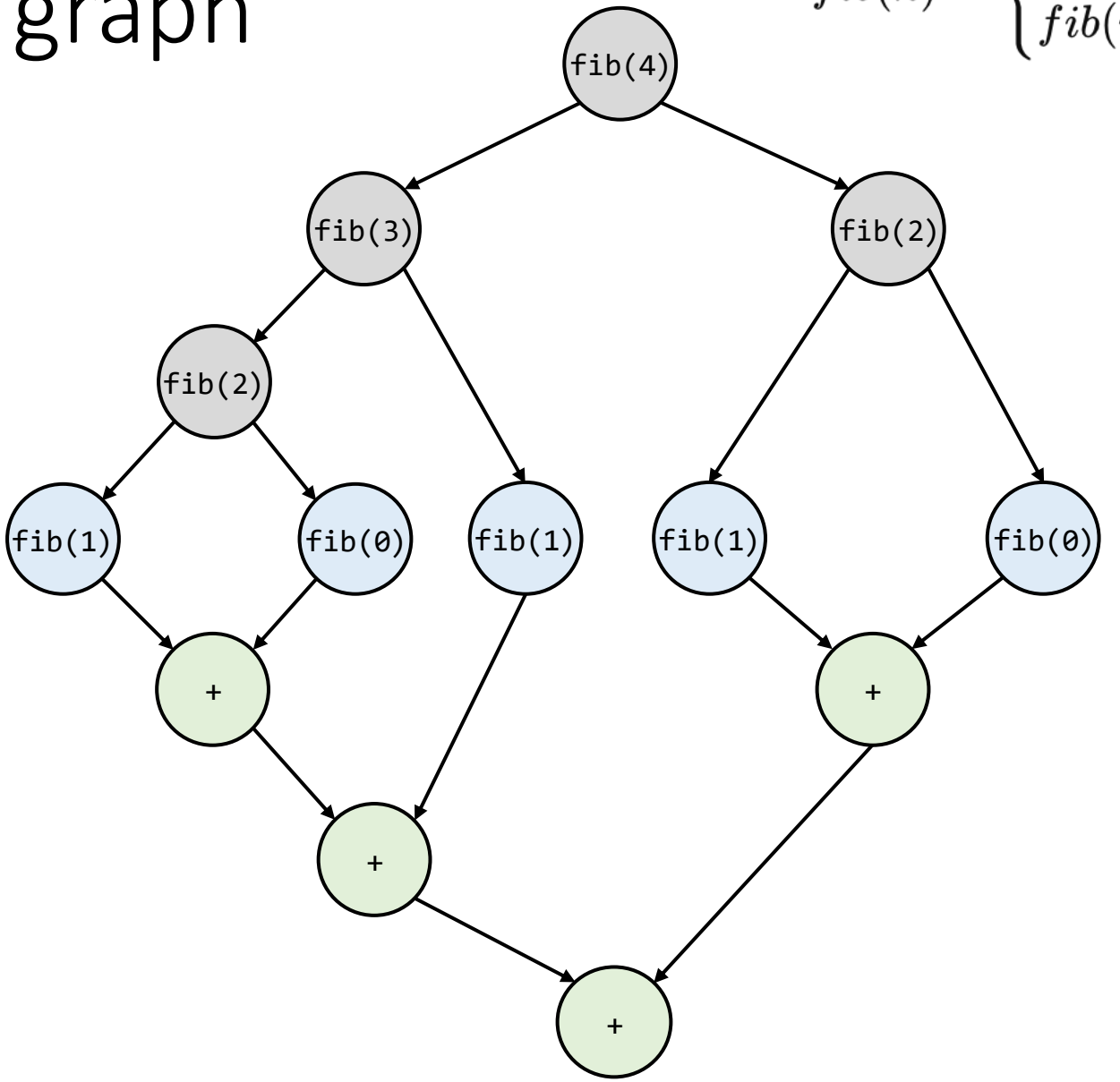
Parallel version (FJ)

```
public class ParallelFib extends RecursiveTask<Integer> {
…
protected Integer compute {
    if (n <= 1) return n;
    ParallelFib f1 = new ParallelFib(n - 1);
    ParallelFib f2 = new ParallelFib(n - 2);

    f1.fork();              // Fork the first subproblem
    int y = f2.compute(); // Compute the second subproblem in
                             the current thread
    int x = f1.join();     // Join the first subproblem's result
    return x + y;
}
```

fib(4) task graph

$$fib(n) = \begin{cases} n, & \text{if } n \leq 1 \\ fib(n-1) + fib(n-2), & \text{if } n > 1 \end{cases}$$



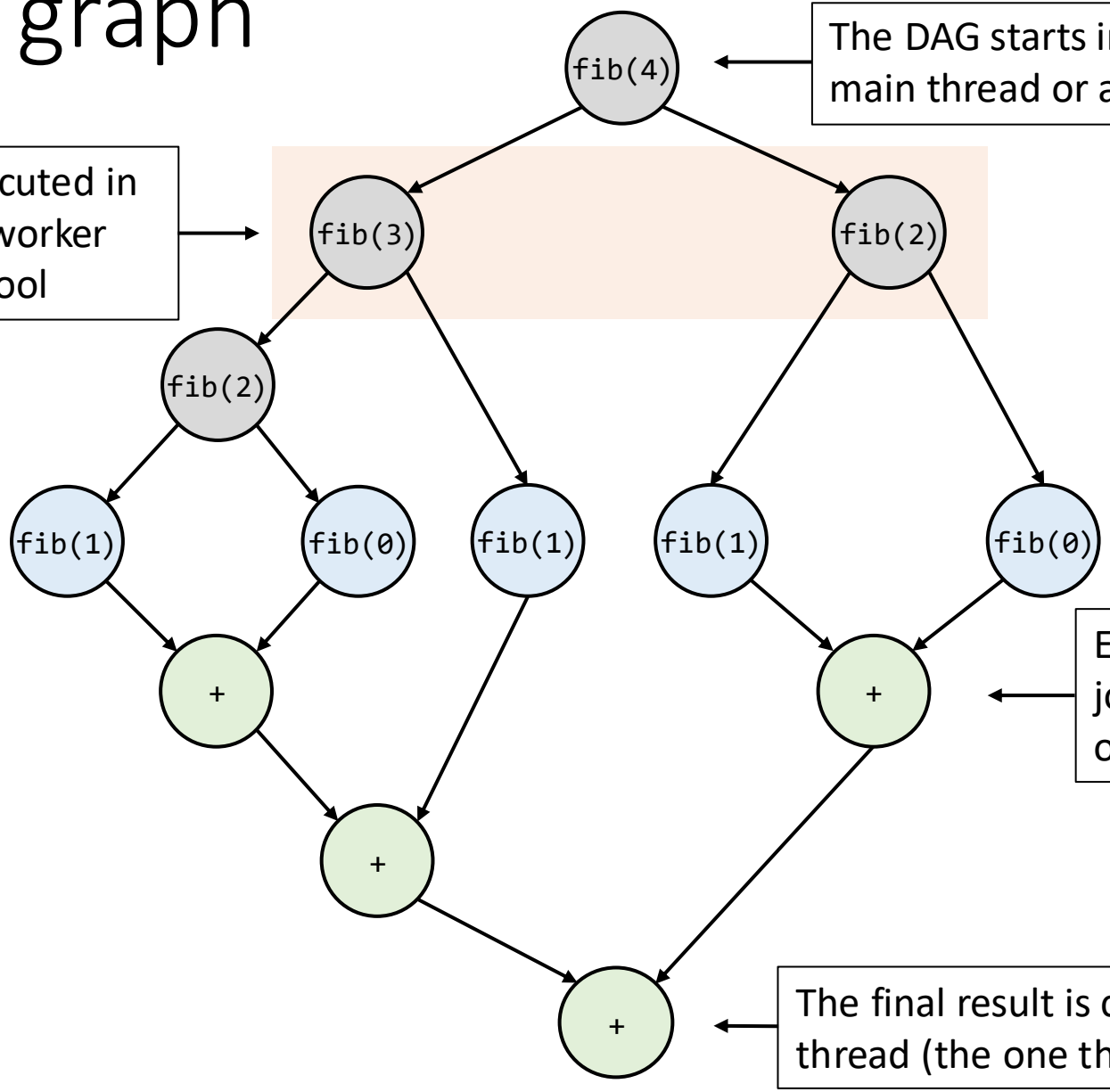

```
if (n <= 1) return n;

…f1 = new …(n - 1);
…f2 = new …(n - 2);

f1.fork();
int y = f2.compute();
int x = f1.join();

return x + y;
```

fib(4) task graph



The DAG starts in a single thread (e.g., the main thread or a worker thread)

Tasks are forked and executed in parallel across multiple worker threads in the ForkJoinPool

Each forked task eventually joins (synchronizing with the original computation)

The final result is computed and returned in a single thread (the one that initiated the computation)

From DAG to execution time

The DAG captures task dependencies and potential parallelism but does not tell us actual execution time

Recall:

- **T_1 (work):** The total execution time if only one thread runs everything sequentially
- **T_p (parallel time):** The execution time when running on P worker threads
- **T_∞ (span):** The execution time assuming infinite threads (pure dependency bottleneck)

From DAG to execution time

The DAG captures task dependencies and potential parallelism but does not tell us actual execution time

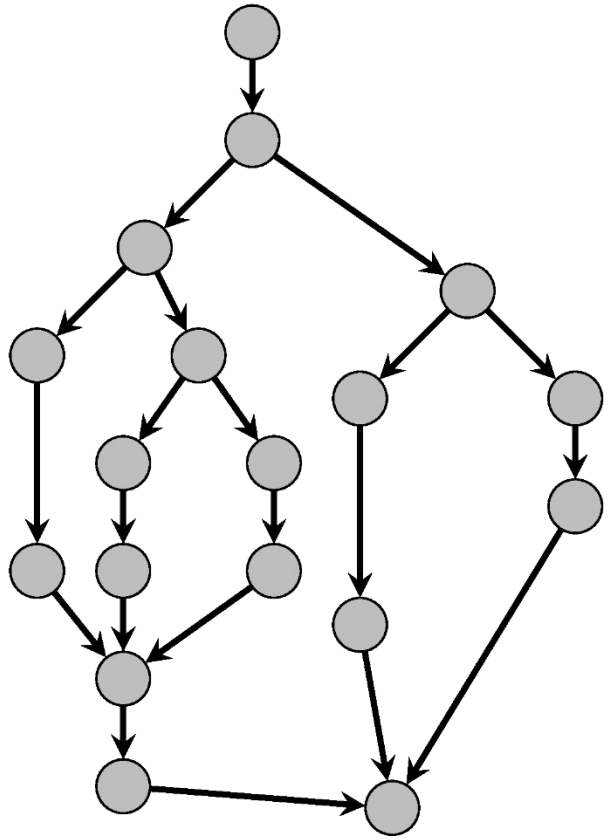
Recall:

- **T_1 (work):** **Tells us total work** - the sum of all task execution times
- **T_p (parallel time):** **Shows real-world parallel performance** - how much speedup we get with P threads
- **T_∞ (span):** **Tells us the "best possible speedup"** - the longest dependency chain that cannot be parallelized

Task parallelism: Performance model

T_1: work (total amount of work)

- The sum of the time cost of all nodes in graph
- As if we executed graph sequentially (p=1)

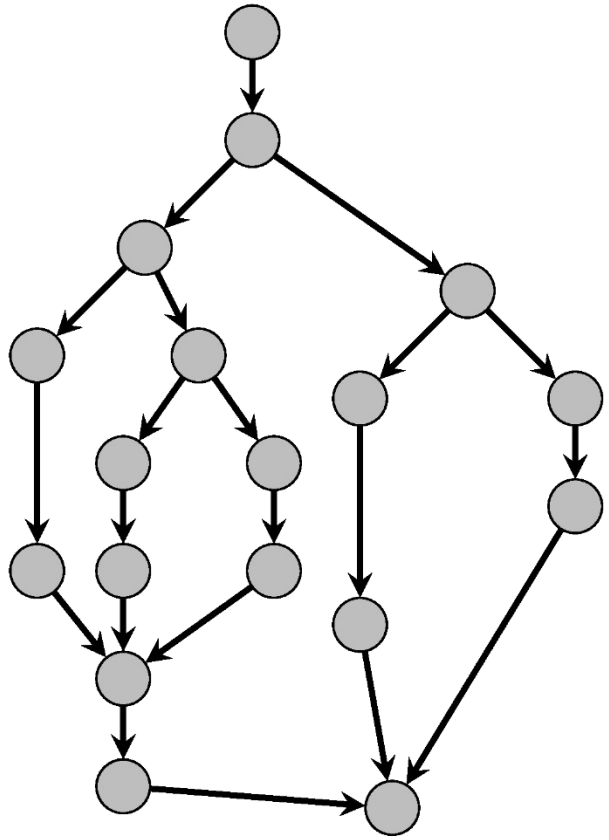


On this graph, T_1 is 18

Task parallelism: Performance model

T_p: execution time on **p** threads

- We can control p by configuring the thread pool (ForkJoinPool(p))
- We cannot control T_p - it depends on:
  - **Task dependencies in the DAG**
  - **Work distribution and load balancing**
  - **Thread contention and scheduling by the JVM/OS**



Task parallelism: Performance model

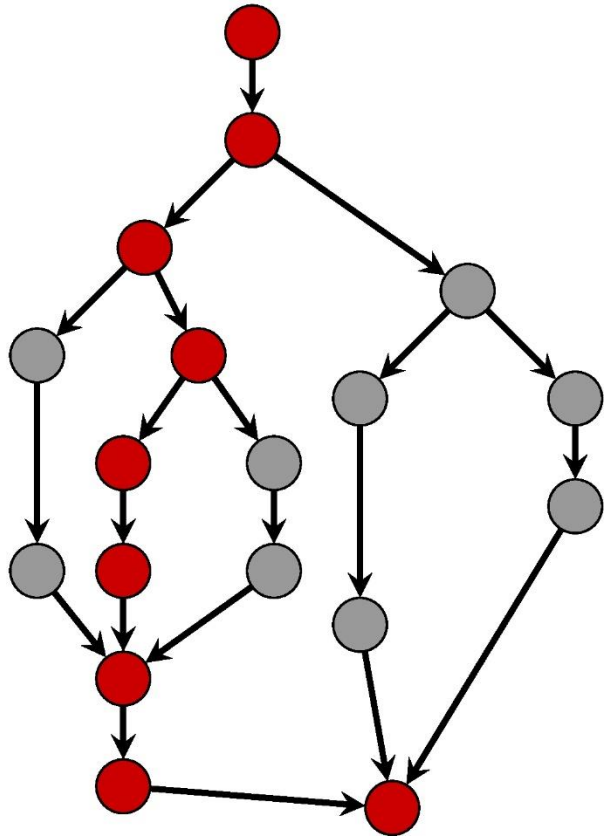
Speedup $S_p = T_1 / T_p$

T_∞: span, critical path length

- Longest chain of dependent operations in the DAG
- Best possible time using infinite cores

Parallelism = T_1 / T_∞

- How much parallelism exist in the task graph
- Gives an upper bound on how much speedup we can get



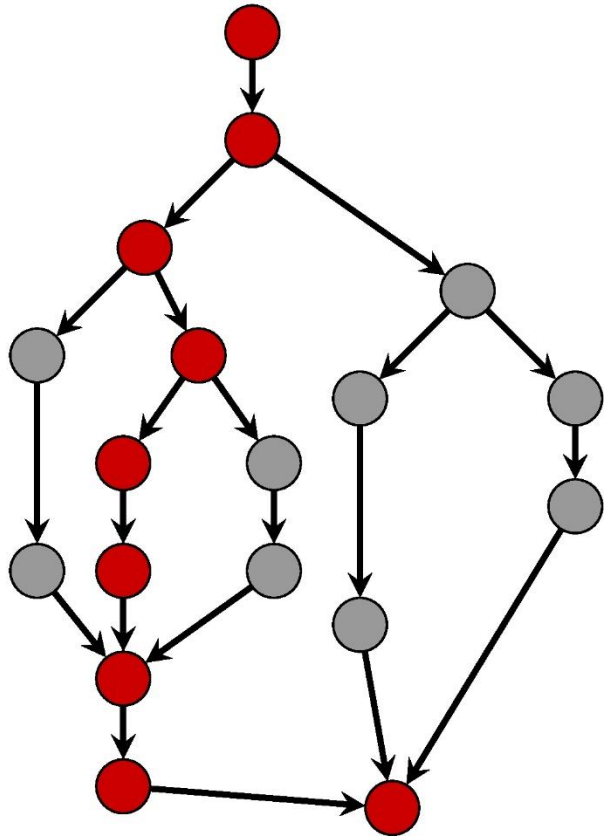
On this graph, T_∞ is 9

Task parallelism: Performance model

On this graph: $T_1 / T_\infty = 18 / 9 = 2$

This means that even with infinite cores and threads, the **best possible speedup is 2**

Bottleneck is the span – DAG structure does not allow more parallelism



Intuition from the DAG

A wide task graph → higher potential parallelism (shorter T_∞)

A deep task graph → more sequential dependencies (longer T_∞, less speedup)

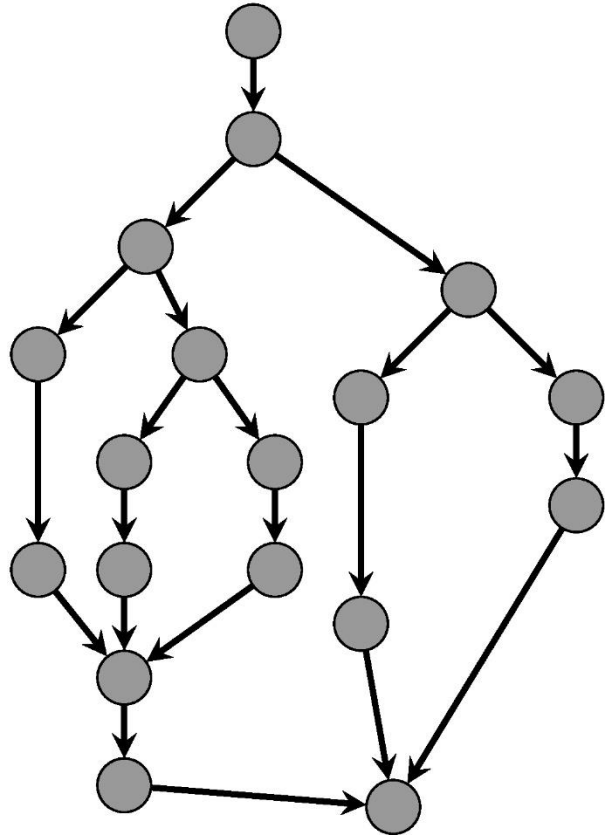
Speedup is limited by the ratio T_1 / T_∞ (work versus critical path → Amdahl's law)

Parallelism is limited by dependencies

Scheduling of task graphs

Performance depends not just on the work and span, the tasks must be scheduled efficiently

Scheduler is an algorithm for assigning **tasks** to **worker threads** (and cores)

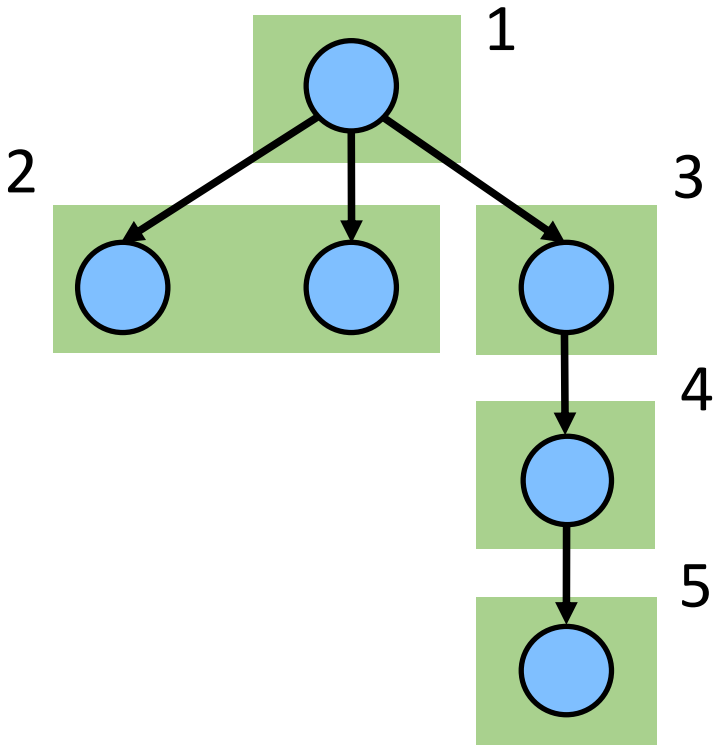


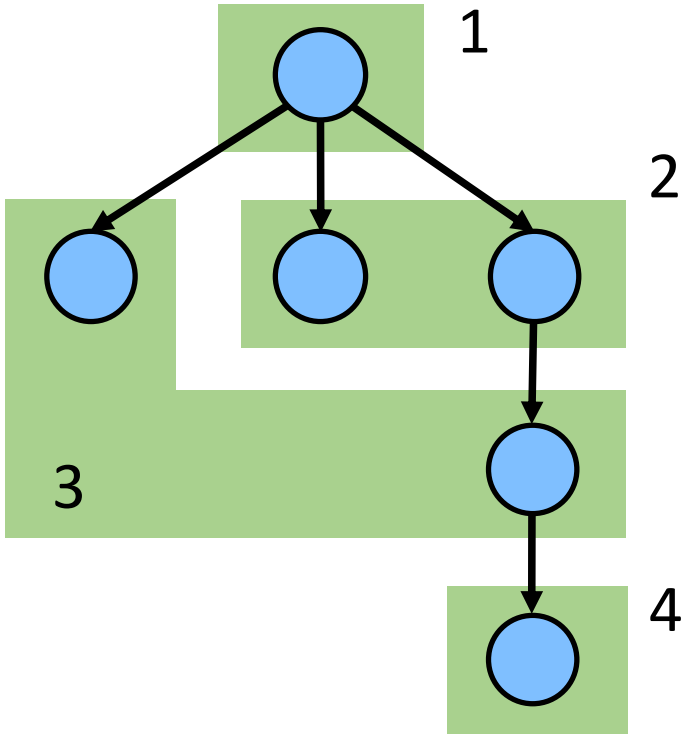
Note that:

T_p depends on scheduler

T_1 , T_∞ , p depends on programmer

What is T2 for this graph?





T_2 will be 5 with this scheduling
(we have 5 time steps)

T_2 will be 4 with this scheduling
(we have 4 time steps)

DAG bounds to execution time

Lower bound:

- Work law: $\mathbf{T_p \geq T_1 / p}$
- Span law: $\mathbf{T_p \geq T_\infty}$
- $\mathbf{T_p \geq max(T_1 / p, T_\infty)}$ -> no scheduler can do better than this

Upper bound:

- Parallelizable work: $\mathbf{T_1 / p}$
- DAG dependencies: $\mathbf{O(T_\infty)}$
- $\mathbf{T_p \leq T_1 / p + O(T_\infty)}$

Near-optimal T_p: Thanks ForkJoin library!

- Inspired by the Cilk runtime system, Java Fork-Join scheduler adopts similar work-stealing techniques to achieve efficient task distribution and low overhead
- ForkJoin framework attempts to make T_p approach this bound by
  - Ensuring good load balancing (T_1/p is not artificially inflated)
  - Reducing scheduling overhead (to prevent T_p from significantly exceeding T_{∞})
- The ForkJoin Framework gives an expected-time guarantee of **asymptotically optimal**!
- FJ work stealing scheduler: $\mathbf{T_p = O(T_1 / p + T_{\infty})}$

Division of responsibility

The **framework-writer**'s job:

- Assign work to available threads to avoid idling
  - Let framework-user ignore all scheduling issues
- Give the expected-time optimal guarantee assuming framework-user did his/her job

The **programmer**'s job as ForkJoin Framework user:

- Make all the nodes a small-ish and approximately equal amount of work
- Designing parallel algorithms is about decreasing span without increasing work too much

Designing parallel algorithms

Designing parallel algorithms is about decreasing span without increasing work too much

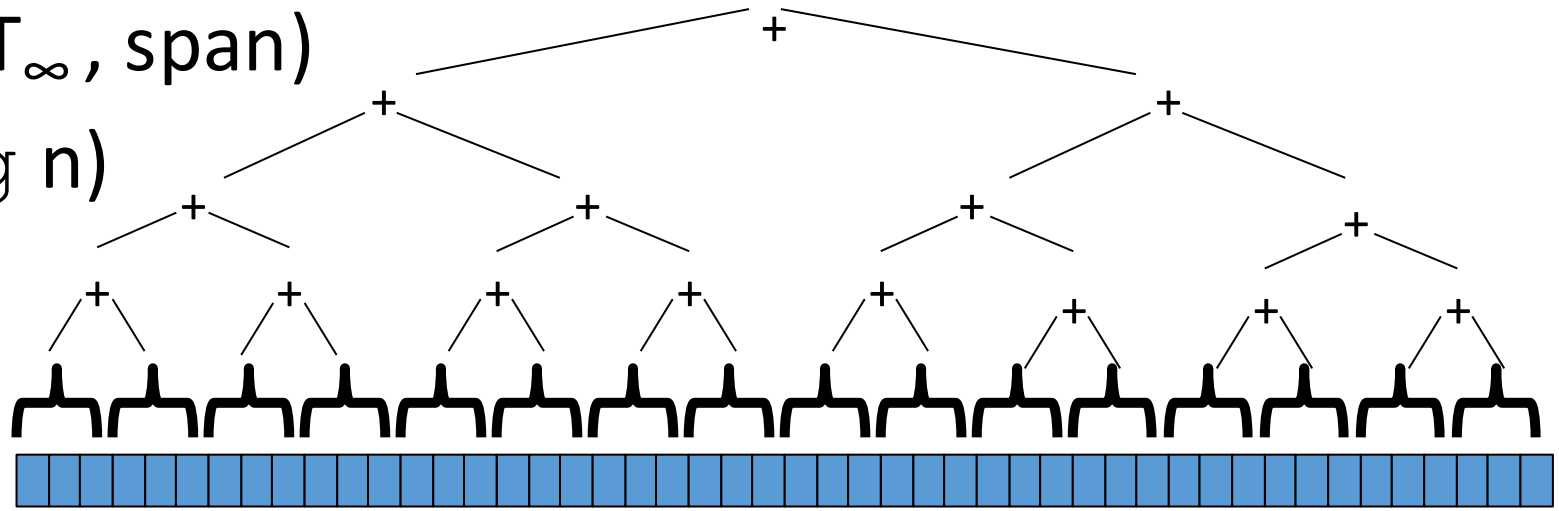
- Amdahl's Law describes the limit of speedup due to sequential parts of a program
- T_∞ (span) in the DAG is the practical representation of the "sequential fraction" in Amdahl's Law
- T_∞ is the fundamental cause of the speedup limit - it represents the longest sequential dependency
- If we reduce T_∞, we get closer to ideal speedup

Parallel patterns

Asymptotic bound

Summing an array:

- Sequential: O(n) (= T1)
- Parallel: O (`log` n) (=T_∞, span)
- Parallelism: O(n / `log` n)



Anything that can use results from two halves and merge them in O(1) time has the same property…

Divide-and-Conquer: What else looks like this?

Maximum or minimum element

Is there an element satisfying some property (e.g., is there a 17)?

Left-most element satisfying some property (e.g., first 17)

- What should the recursive tasks return?
- How should we merge the results?

Counts, for example, number of strings that start with a vowel

- This is just summing with a different base case
- Many problems are!

Reductions

Computations of this form are called **reductions**

Produce single answer from collection via an **associative operator**

- Examples: max, count, leftmost, rightmost, sum, product, …
- Non-examples: median, subtraction, exponentiation

(Recursive) results don't have to be single numbers or strings. They can be arrays or objects with multiple fields.

- Example: Histogram of test results is a variant of sum

But some things are inherently sequential

- How we process `arr[i]` may depend entirely on the result of processing `arr[i-1]`

Maps

Standard **map** applies a function to each element of a collection, producing an output of the same size

- Example: Squaring each element of an array
  `B[i] = f(A[i])`

Extending the concept: what if the function takes multiple inputs?

- Zip map (element-wise map): applies a function to corresponding elements from multiple collections
- Output has still the same size as the input collection
- Example: Element-wise addition of two arrays

Code example: PP-L11-02MapsArraySummation

Maps in ForkJoin Framework (compute)

```
protected void compute(){
if(hi-lo < SEQUENTIAL_CUTOFF) {
    for(int i = lo; i < hi; i++)
        res[i] = arr1[i] + arr2[i]; // 2 inputs, 1 output
    } else {
    int mid = (hi + lo) / 2;
    VecAdd left = new VecAdd(lo, mid, res, arr1, arr2);
    VecAdd right= new VecAdd(mid, hi, res, arr1, arr2);
    left.fork();
    right.compute();
    left.join();
    }
}
```

Maps in ForkJoin Framework (use)

```
class Globals {
    static ForkJoinPool fjPool = new ForkJoinPool();
}


static int[] add(int[] arr1, int[] arr2){
    assert (arr1.length == arr2.length);
    int[] ans = new int[arr1.length];
    Globals.fjPool.invoke(new VecAdd(0,ans.length,ans,arr1,arr2));
    return ans;
}
```

Maps and reductions

Maps and reductions: the “work horses” of parallel programming

- By far the two most important and common patterns
- Learn to recognize when an algorithm can be written in terms of maps and reductions
- Use maps and reductions to describe (parallel) algorithms
  - **Reduction always achieve O(log n) parallel time**
  - **Map achieves O(log n) with Divide & Conquer**
- Programming them becomes “trivial” with a little practice

Digression: MapReduce on clusters

You may have heard of Google's "map/reduce"

- Or the open-source version Hadoop

Idea: Perform maps/reduces on data using many machines

- The system takes care of distributing the data and managing fault tolerance
- You just write code to map one element and reduce elements to a combined result

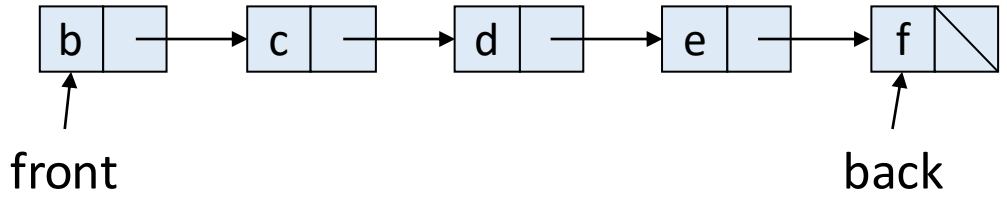
Separates how to do recursive divide-and-conquer from what computation to perform

- Old idea in higher-order functional programming transferred to large-scale distributed computing

Linked lists

Can you parallelize maps or reduces over linked lists?

- Example: Increment all elements of a linked list
- Example: Sum all elements of a linked list
- Parallelism still beneficial for expensive per-element operations



- Once again, data structures matter!
- **For parallelism, balanced trees generally better than lists so that we can get to all the data exponentially faster O(log n) vs. O(n)**
  - Trees have the same flexibility as lists compared to arrays

Amdahl's Law (mostly bad news)

So far: analyze parallel programs in terms of work and span

In practice, typically have parts of programs that parallelize well…

- Such as maps/reductions over arrays and trees

…and parts that don't parallelize at all

- Such as reading a linked list, getting input, doing computations where each needs the previous step, etc.

All is not lost

Amdahl's Law is a bummer!

- Sequential parts become a bottleneck very quickly
- But it doesn't mean additional processors are worthless

We can find new parallel algorithms

- Some things that seem inherently sequential are actually parallelizable
- Rewrite it in a way that allows parallelization

Parallel algorithms

The prefix-sum problem

Given **`int[] input`**,

produce **`int[] output`** where:

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

Sequential prefix-sum

```
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

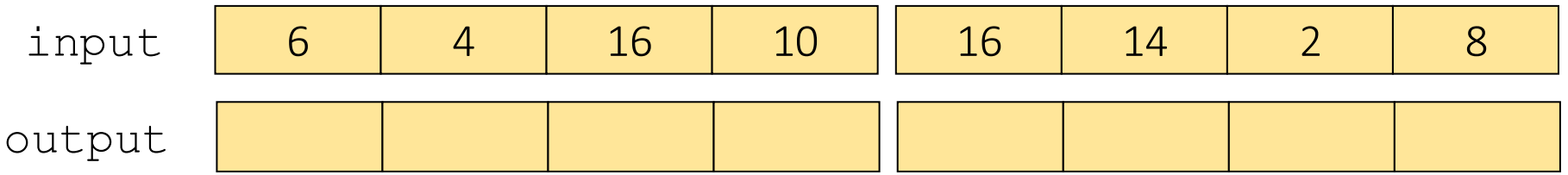
- Work: O(n), Span: O(n)
- This algorithm is sequential, but a **different algorithm** has: Work O(n), Span O(`log` n)

Example

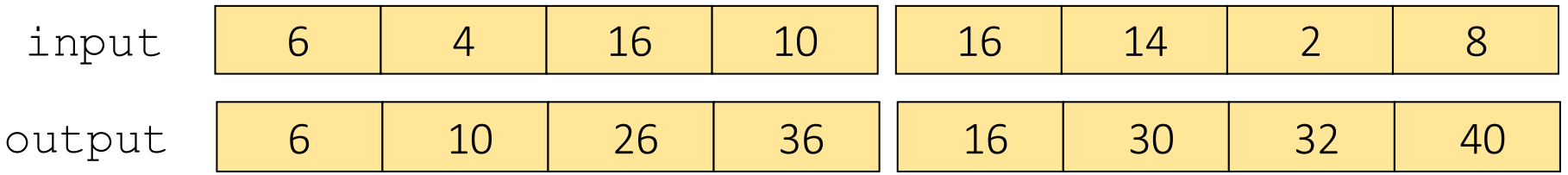
How to compute the prefix-sum in parallel?

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

Example



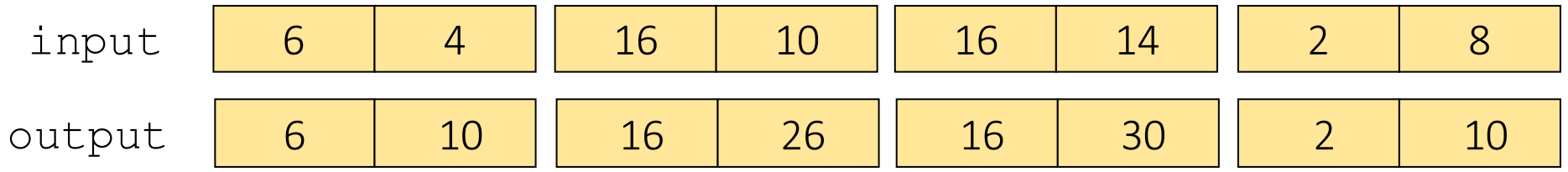
Example



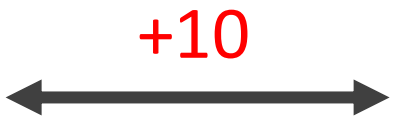
Example

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

Example



Example

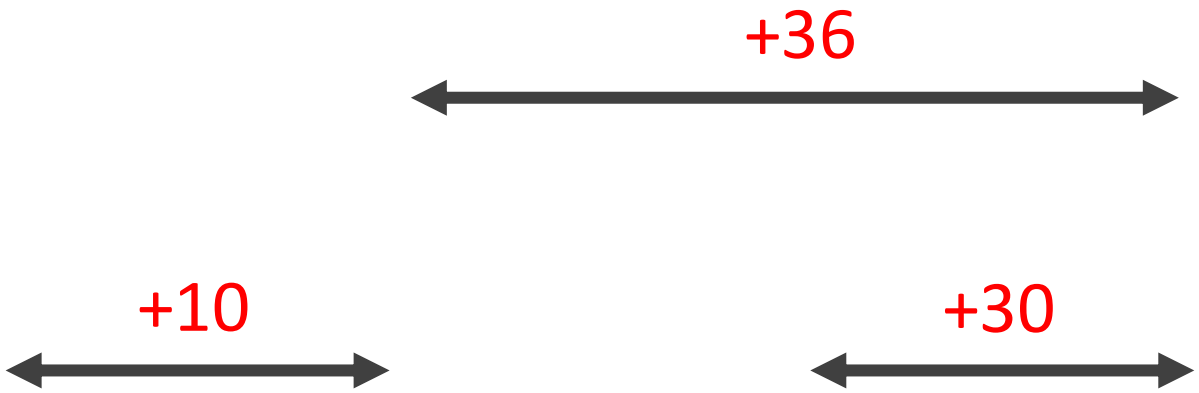


input	6	4	16	10	16	14	2	8
output	6	10	26	36	16	30	2	10

Example

input	6	4	16	10	16	14	2	8
output	6	10	26	36	16	30	32	40

Example



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

Parallel prefix-sum

The parallel-prefix algorithm does two passes

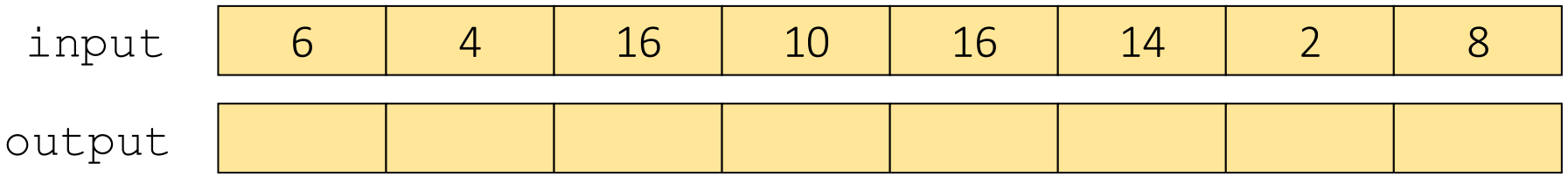
- Each pass has O(n) work and O(`log` n) span
- So in total there is O(n) work and O(`log` n) span
- So like with array summing, parallelism is n/`log` n

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

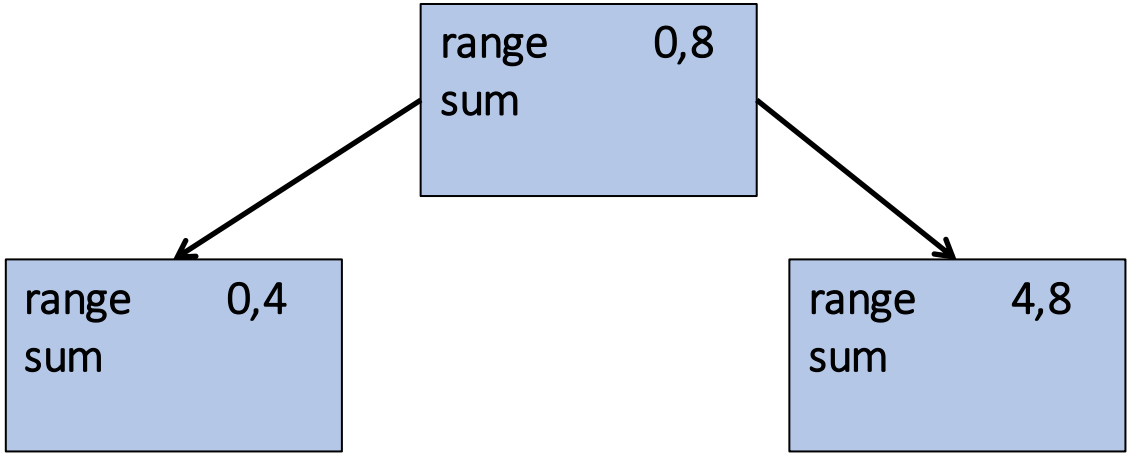
Second pass traverses the tree top-down: the “down” pass

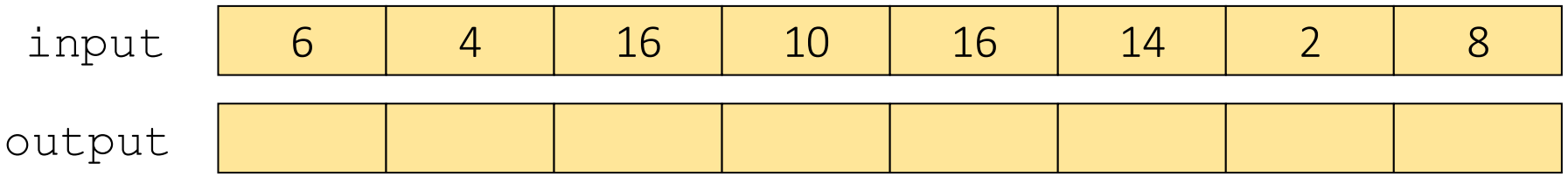
Example

range	0,8
sum	

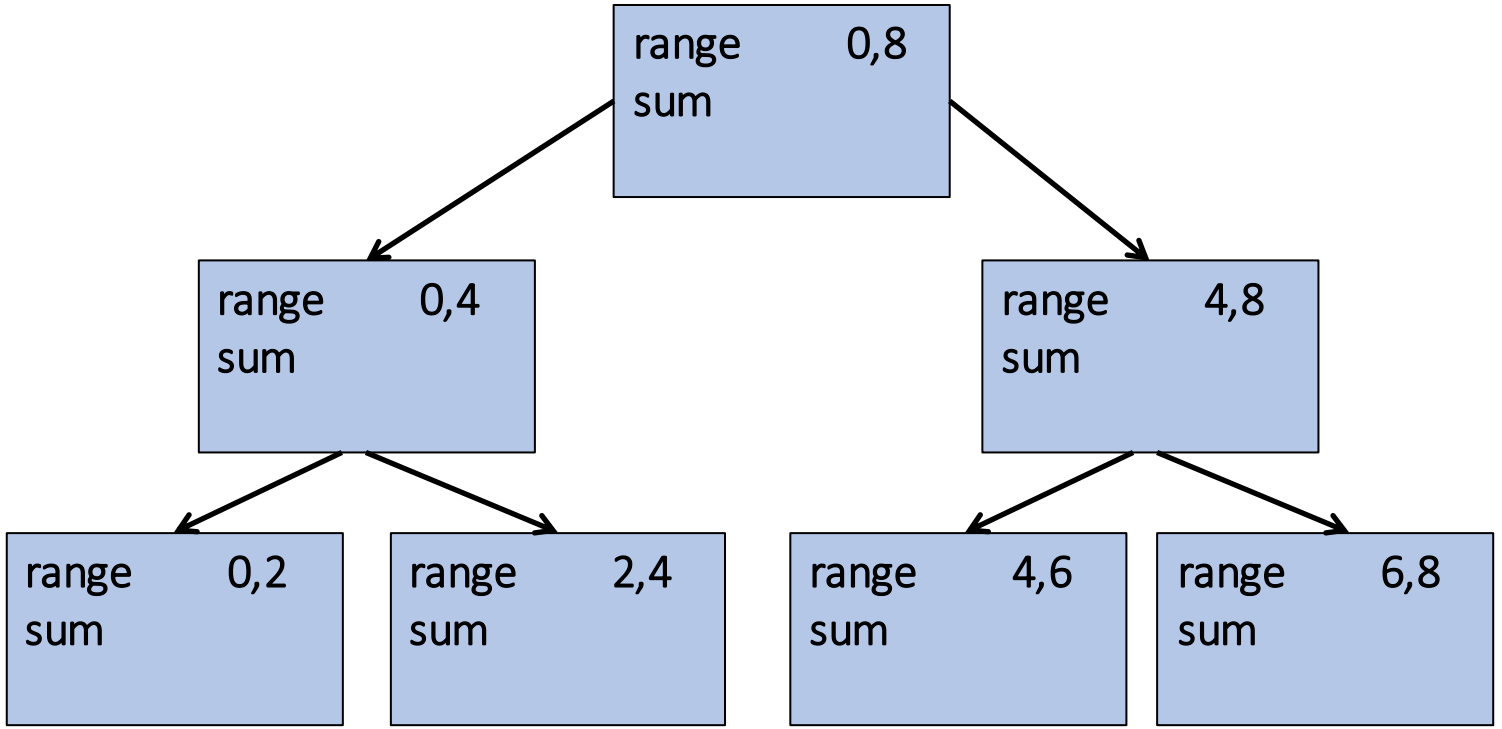


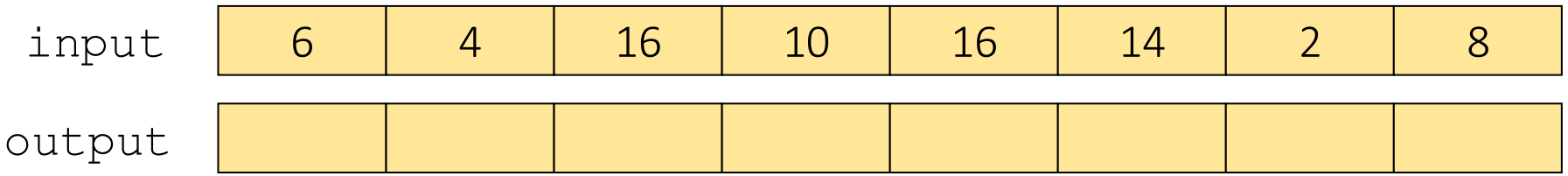
Example


range 0,8
sum
range 0,4
sum
range 4,8
sum

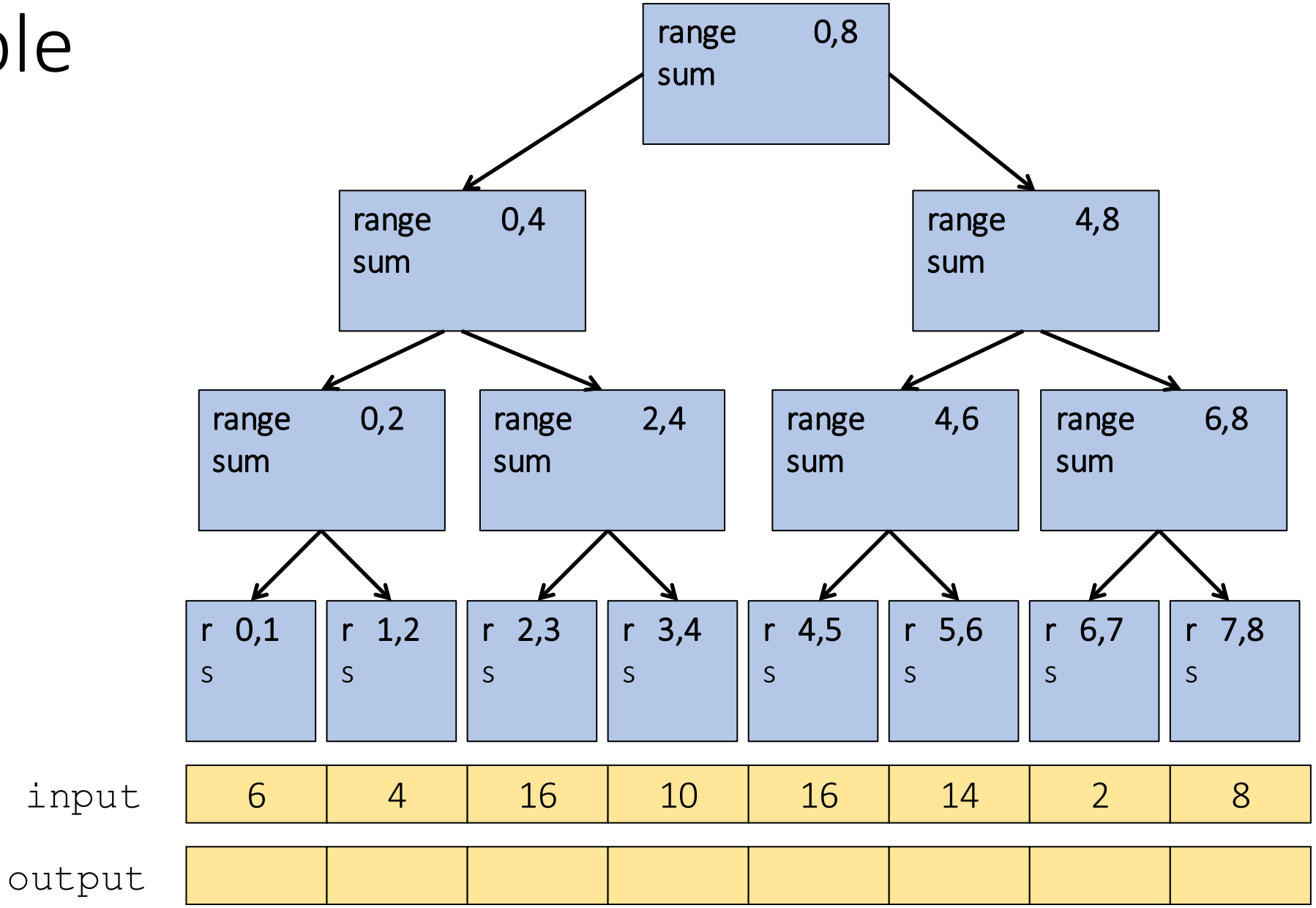

input
6
4
16
10
16
14
2
8
output

Example

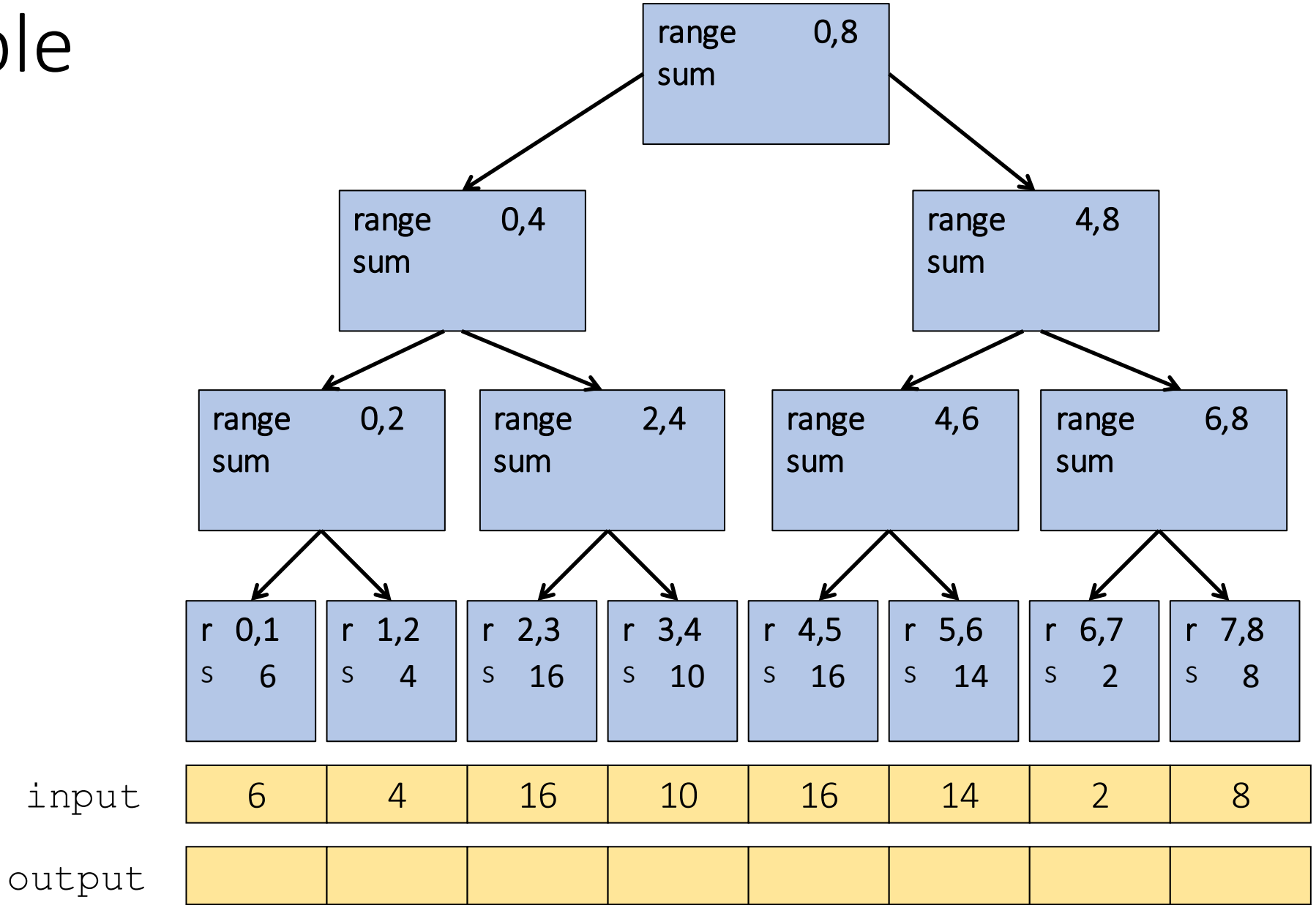

range 0,8
sum
range 0,4
sum
range 4,8
sum
range 0,2
sum
range 2,4
sum
range 4,6
sum
range 6,8
sum


input
6
4
16
10
16
14
2
8
output

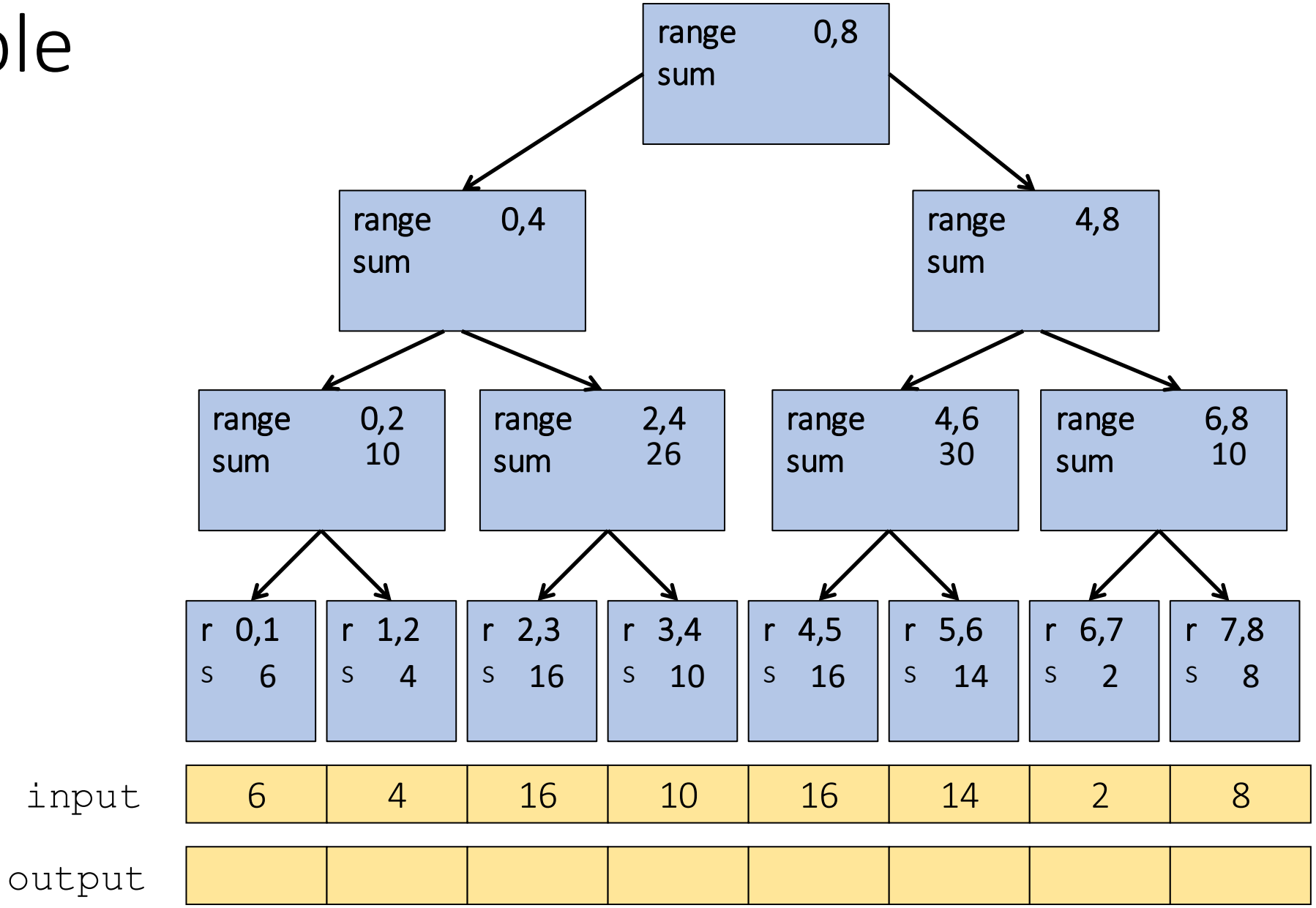
Example


range 0,8
sum
range 0,4
sum
range 4,8
sum
range 0,2
sum
range 2,4
sum
range 4,6
sum
range 6,8
sum
r 0,1
s
r 1,2
s
r 2,3
s
r 3,4
s
r 4,5
s
r 5,6
s
r 6,7
s
r 7,8
s
input
6
4
16
10
16
14
2
8
output

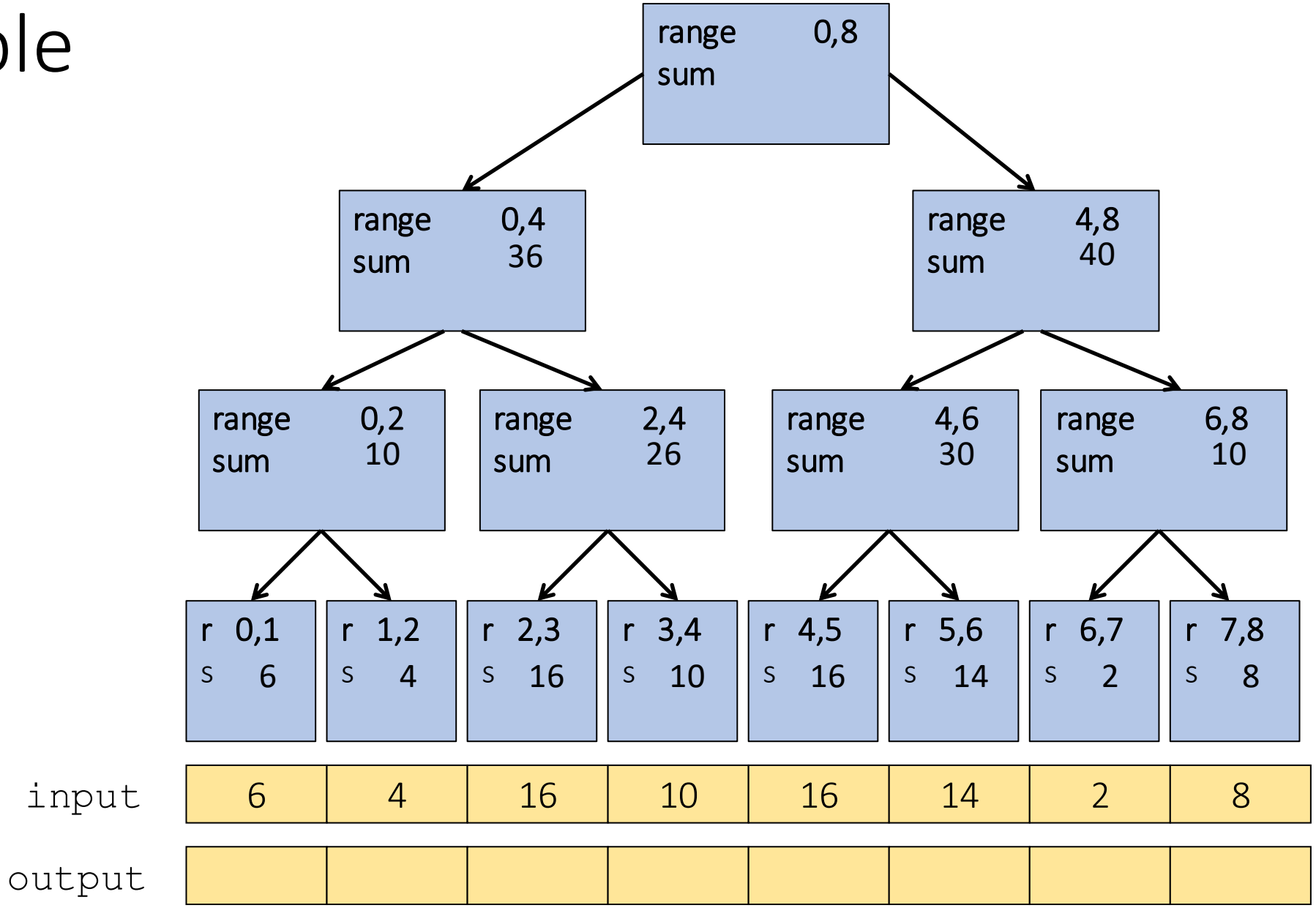
Example


range 0,8
sum
range 0,4
sum
range 4,8
sum
range 0,2
sum
range 2,4
sum
range 4,6
sum
range 6,8
sum
r 0,1
s 6
r 1,2
s 4
r 2,3
s 16
r 3,4
s 10
r 4,5
s 16
r 5,6
s 14
r 6,7
s 2
r 7,8
s 8
input
6
4
16
10
16
14
2
8
output

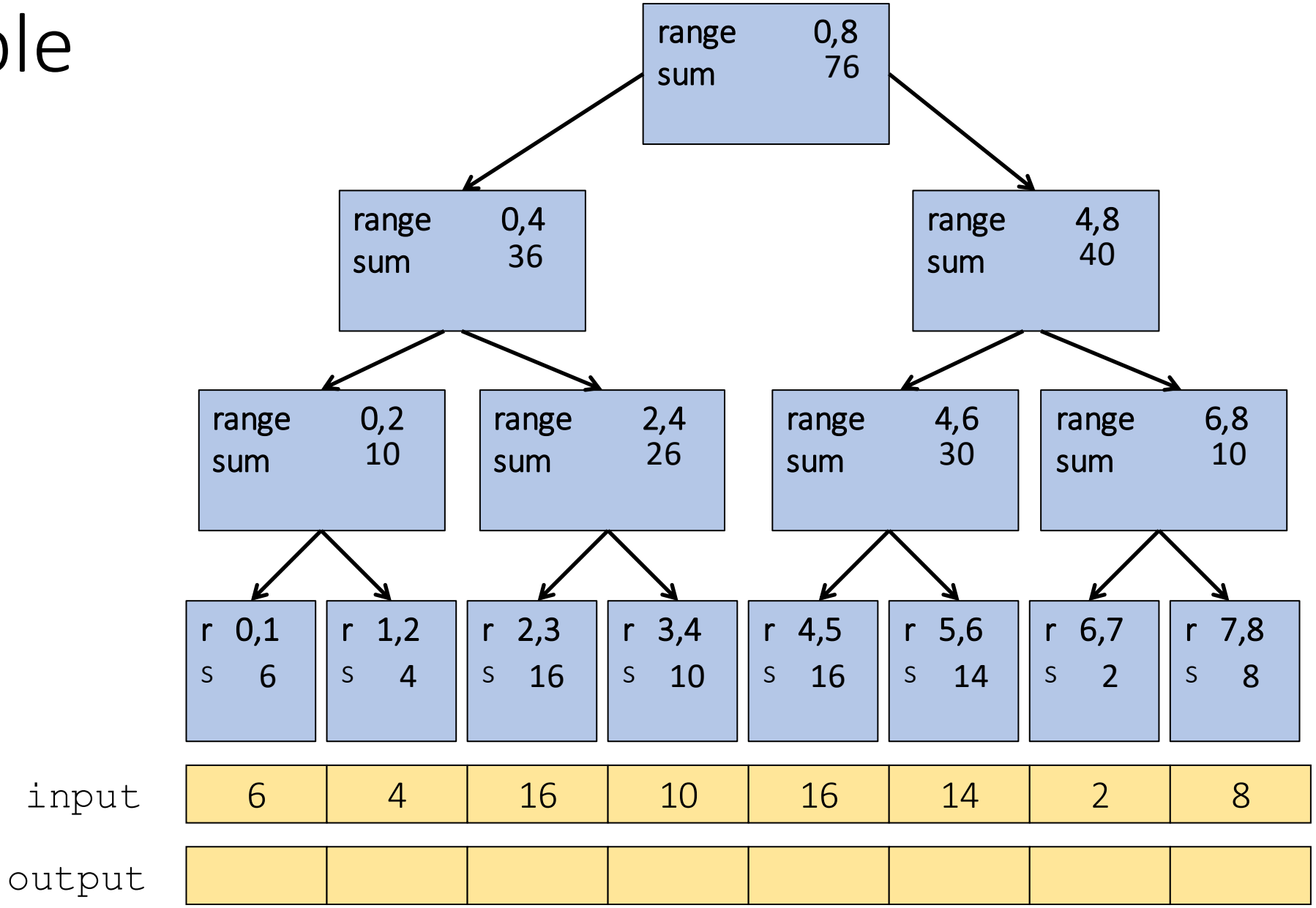
Example


range 0,8
sum
range 0,4
sum
range 4,8
sum
range 0,2
sum 10
range 2,4
sum 26
range 4,6
sum 30
range 6,8
sum 10
r 0,1
s 6
r 1,2
s 4
r 2,3
s 16
r 3,4
s 10
r 4,5
s 16
r 5,6
s 14
r 6,7
s 2
r 7,8
s 8
input
6
4
16
10
16
14
2
8
output

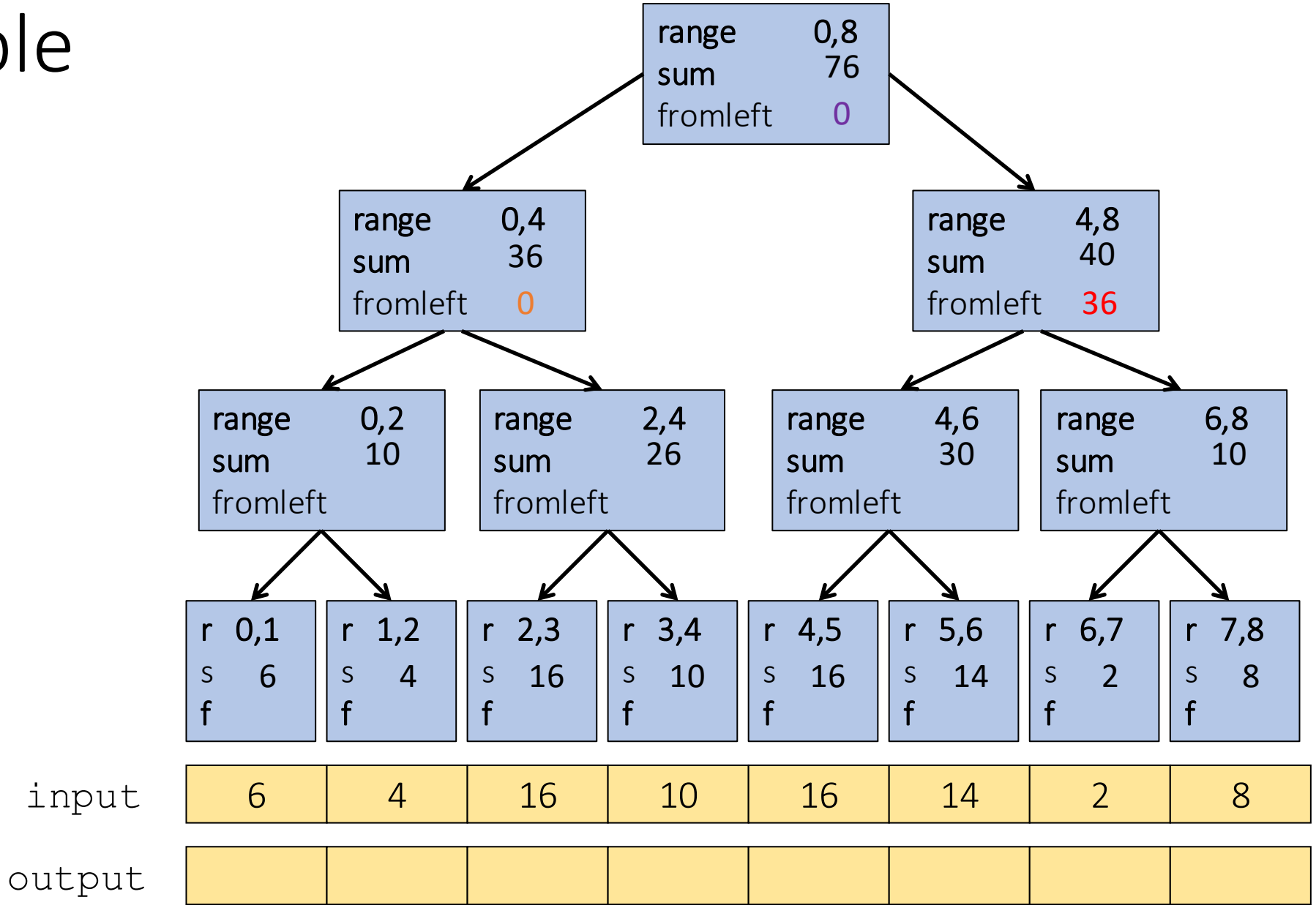
Example


range 0,8
sum
range 0,4
sum 36
range 4,8
sum 40
range 0,2
sum 10
range 2,4
sum 26
range 4,6
sum 30
range 6,8
sum 10
r 0,1
s 6
r 1,2
s 4
r 2,3
s 16
r 3,4
s 10
r 4,5
s 16
r 5,6
s 14
r 6,7
s 2
r 7,8
s 8
input
6
4
16
10
16
14
2
8
output

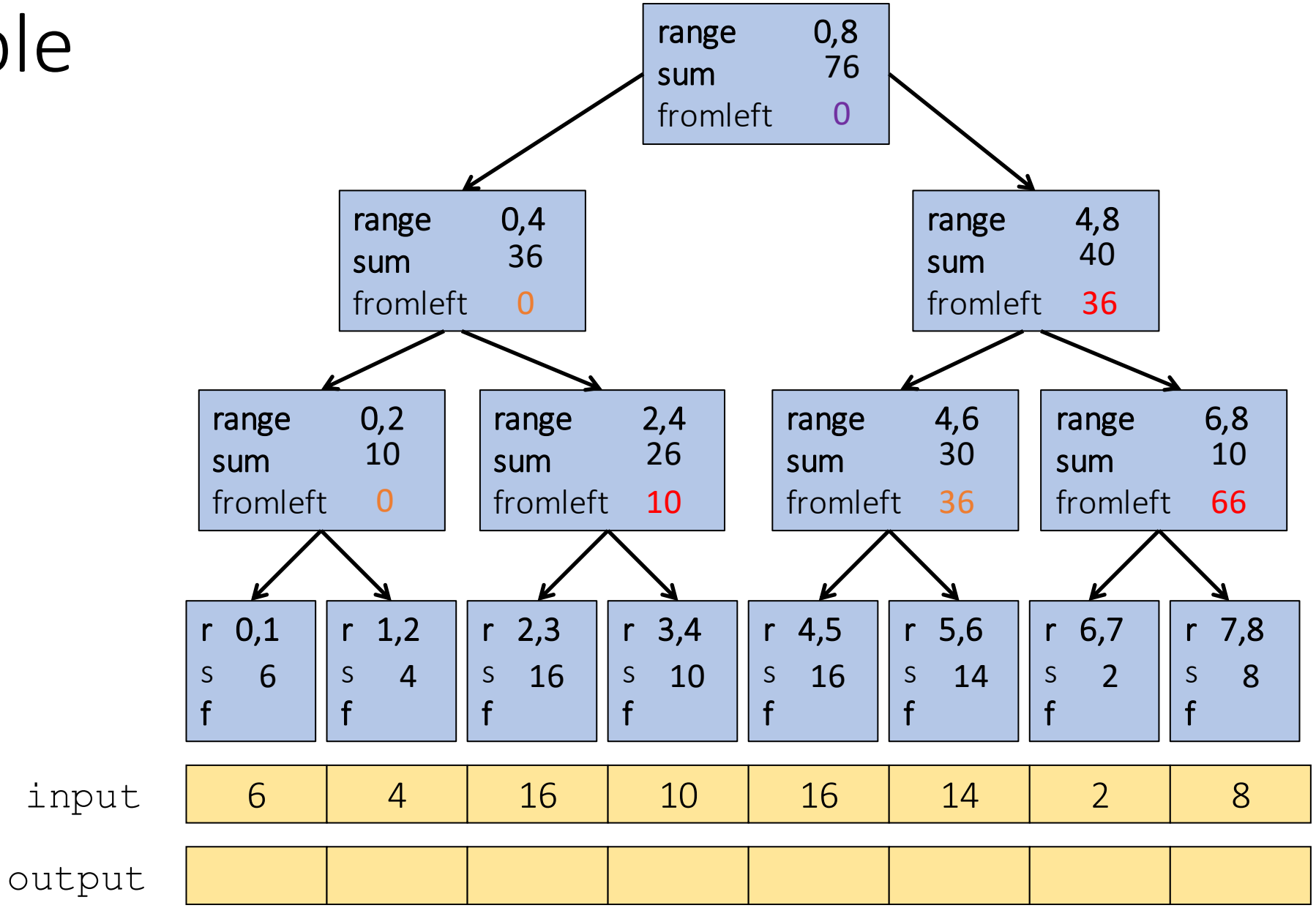
Example


range 0,8
sum 76
range 0,4
sum 36
range 4,8
sum 40
range 0,2
sum 10
range 2,4
sum 26
range 4,6
sum 30
range 6,8
sum 10
r 0,1
s 6
r 1,2
s 4
r 2,3
s 16
r 3,4
s 10
r 4,5
s 16
r 5,6
s 14
r 6,7
s 2
r 7,8
s 8
input
6
4
16
10
16
14
2
8
output

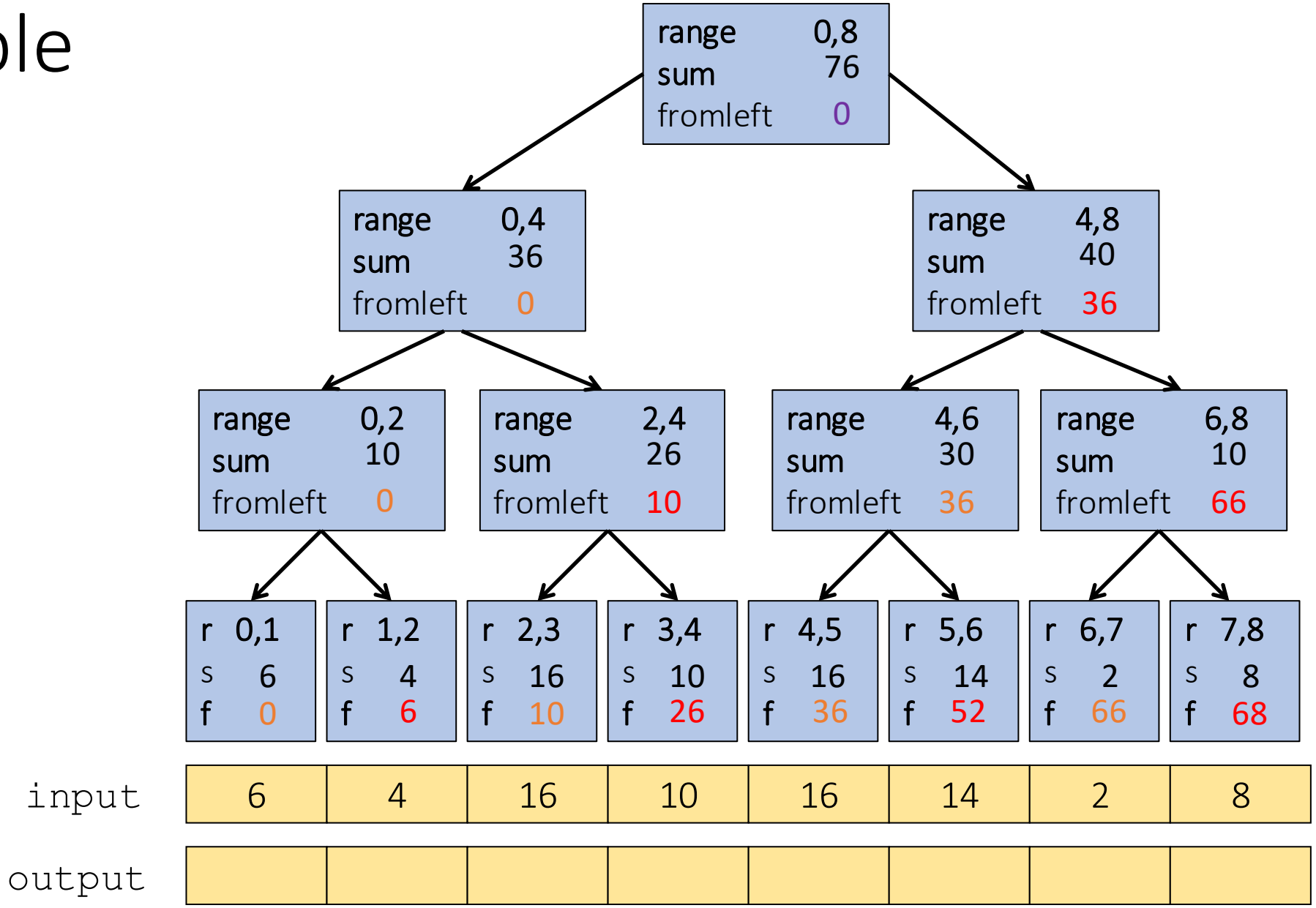
Example


range 0,8
sum 76
fromleft 0
range 0,4
sum 36
fromleft 0
range 4,8
sum 40
fromleft 36
range 0,2
sum 10
fromleft
range 2,4
sum 26
fromleft
range 4,6
sum 30
fromleft
range 6,8
sum 10
fromleft
r 0,1
s 6
f
r 1,2
s 4
f
r 2,3
s 16
f
r 3,4
s 10
f
r 4,5
s 16
f
r 5,6
s 14
f
r 6,7
s 2
f
r 7,8
s 8
f
input
6
4
16
10
16
14
2
8
output

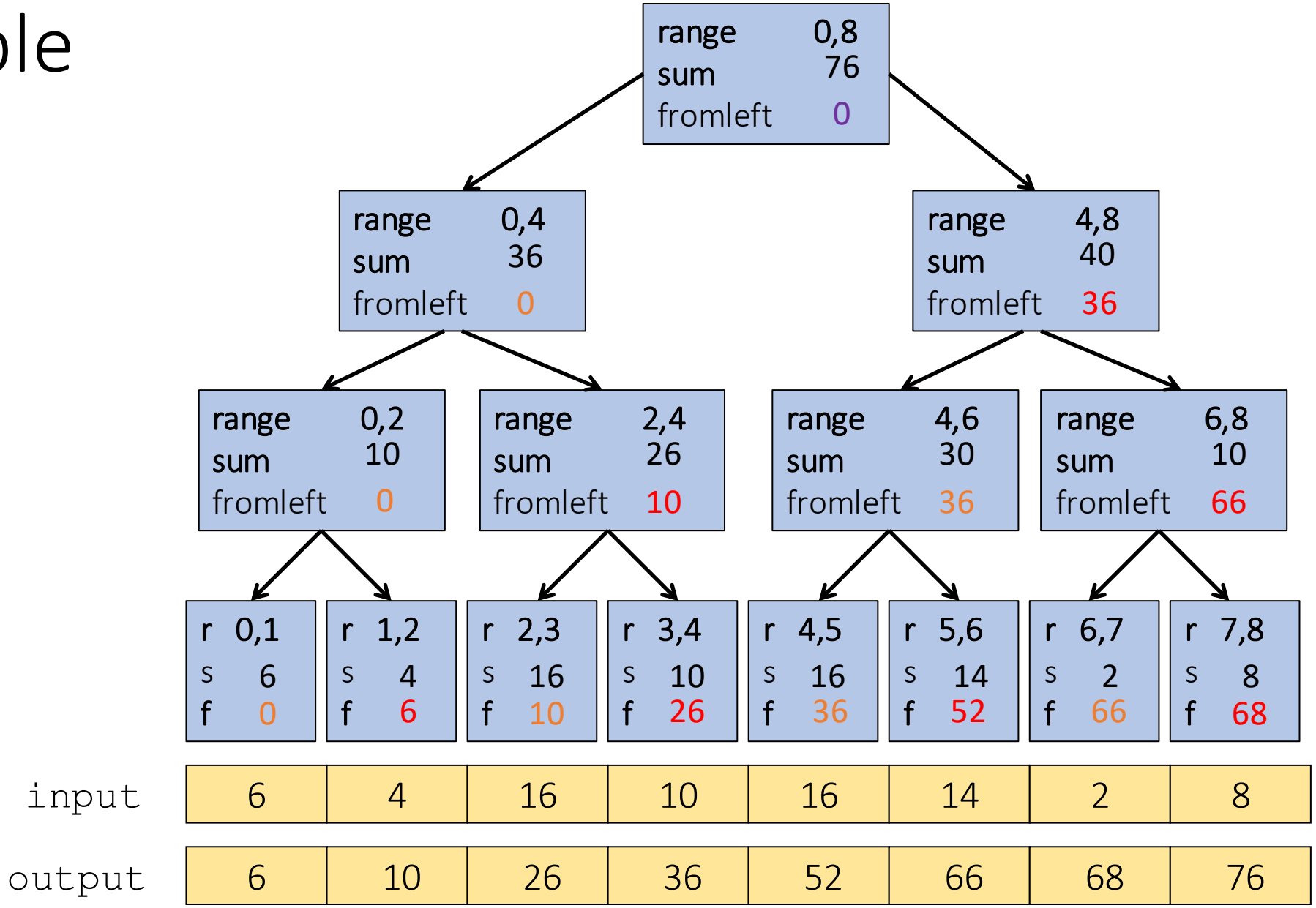
Example


range 0,8
sum 76
fromleft 0
range 0,4
sum 36
fromleft 0
range 4,8
sum 40
fromleft 36
range 0,2
sum 10
fromleft 0
range 2,4
sum 26
fromleft 10
range 4,6
sum 30
fromleft 36
range 6,8
sum 10
fromleft 66
r 0,1
s 6
f
r 1,2
s 4
f
r 2,3
s 16
f
r 3,4
s 10
f
r 4,5
s 16
f
r 5,6
s 14
f
r 6,7
s 2
f
r 7,8
s 8
f
input
6
4
16
10
16
14
2
8
output

Example


range 0,8
sum 76
fromleft 0
range 0,4
sum 36
fromleft 0
range 4,8
sum 40
fromleft 36
range 0,2
sum 10
fromleft 0
range 2,4
sum 26
fromleft 10
range 4,6
sum 30
fromleft 36
range 6,8
sum 10
fromleft 66
r 0,1
s 6
f 0
r 1,2
s 4
f 6
r 2,3
s 16
f 10
r 3,4
s 10
f 26
r 4,5
s 16
f 36
r 5,6
s 14
f 52
r 6,7
s 2
f 66
r 7,8
s 8
f 68
input
6
4
16
10
16
14
2
8
output

Example


range 0,8
sum 76
fromleft 0
range 0,4
sum 36
fromleft 0
range 4,8
sum 40
fromleft 36
range 0,2
sum 10
fromleft 0
range 2,4
sum 26
fromleft 10
range 4,6
sum 30
fromleft 36
range 6,8
sum 10
fromleft 66
r 0,1
s 6
f 0
r 1,2
s 4
f 6
r 2,3
s 16
f 10
r 3,4
s 10
f 26
r 4,5
s 16
f 36
r 5,6
s 14
f 52
r 6,7
s 2
f 66
r 7,8
s 8
f 68
input
6
4
16
10
16
14
2
8
output
6
10
26
36
52
66
68
76

The algorithm, part 1

1. Up: Build a binary tree 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`**), i.e., **`input[i]`**

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

- Tree built bottom-up in parallel

Analysis: O(n) work, O(`log` n) span

The algorithm, part 2

2. Down: Pass down a value **`fromLeft`**
   - 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

- Leaves assign to **`output`**
- Invariant: **`fromLeft`** is sum of elements left of the node's range

Analysis: O(n) work, O(`log` n) span

Sequential cut-off

Adding a sequential cut-off is easy as always:

Up:

just a sum, have leaf node hold the sum of a range

Down:

```
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 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

[Non-standard terminology]

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

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

`f: is elt > 10`

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

Parallelizable?

- Finding elements for the output is easy
- But getting them in the right place seems hard

Parallel prefix to the rescue

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]
```

Parallel prefix to the rescue

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]
```

Parallel prefix to the rescue

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 (with condition) 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++){
  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(`log` n) span

- 2 or 3 passes, but 3 is a constant

Parallelized packs will help us parallelize quicksort…

Quicksort review

Recall Quicksort was sequential, in-place, expected time O(n `log` n)

	Best / expected case *work*
1. Pick a pivot element	O(1)
2. Partition all the data into:	O(n)
A. The elements less than the pivot	
B. The pivot	
C. The elements greater than the pivot	
3. Recursively sort A and C	2T(n/2)

How should we parallelize this?

Quicksort

	Best / expected case *work*
1. Pick a pivot element	O(1)
2. Partition all the data into:	O(n)
A. The elements less than the pivot	
B. The pivot	
C. The elements greater than the pivot	
3. Recursively sort A and C	2T(n/2)

Easy: Do the two recursive calls in parallel

- Work: unchanged of course O(n log n)
- Span: now T(n) = O(n) + 1T(n/2) = O(n)
- So parallelism (i.e., work / span) is O(log n)

Doing better

O(`log` n) speed-up with an infinite number of processors is okay, but a bit underwhelming

- Sort 10^9 elements 30 times faster

Google searches strongly suggest quicksort cannot do better because the partition cannot be parallelized

- The Internet has been known to be wrong ☺
- But we need auxiliary storage (no longer in place)
- In practice, constant factors may make it not worth it, but remember Amdahl's Law

Already have everything we need to parallelize the partition…

Parallel partition (not in place)

Partition all the data into:

A. The elements less than the pivot
B. The pivot
C. The elements greater than the pivot

This is just two packs!

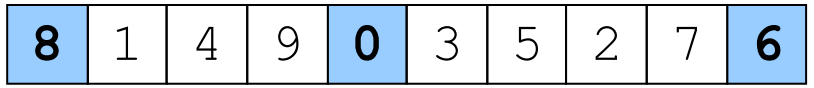
- We know a pack is $O(n)$ work, $O(\log n)$ span
- Pack elements less than pivot into left side of **`aux`** array
- Pack elements greater than pivot into right size of **`aux`** array
- Put pivot between them and recursively sort
- With a little more cleverness, can do both packs at once but no effect on asymptotic complexity

With $O(\log n)$ span for partition, the total span for quicksort is

$T(n) = O(\log n) + 1T(n/2) = O(\log^2 n)$

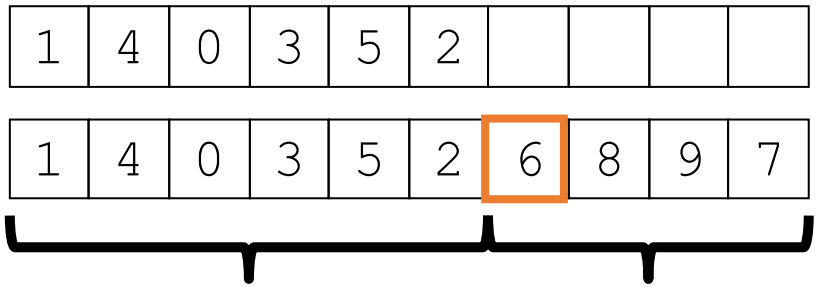
Example

Step 1: pick pivot as median of three



Steps 2a and 2c (combinable): pack less than, then pack greater than into a second array

- Fancy parallel prefix to pull this off not shown



Step 3: Two recursive sorts in parallel

- Can sort back into original array (like in mergesort)

Summary

- FJ spans a DAG → Models dependencies in parallel tasks
- Java ForkJoin (FJ) optimizes T_p → Efficient task scheduling on threads
- Span T_∞ represents sequential dependencies → Sequential parts limit parallelism
- Optimizing T_∞ is on the programmer → Minimize sequential bottlenecks
- Amdahl's Law applies → Reducing sequential execution boosts speedup
- Use parallel patterns → Helps design efficient parallel algorithms, reducing span (independent on programming language)