Parallel Prefix

CSE 332 – Section 8

Slides by James Richie Sulaeman

Parallel Prefix/Suffix

Parallel Pre/suffix

Parallel prefix is a type of programming problem where:

- A given array of elements needs to be **processed in parallel**
- Each element is **combined with its predecessors** through some operation
    - i.e. in parallel prefix sum, each element is summed with its predecessors

Allows for efficient computation of certain operations on large datasets since it enables multiple threads to work simultaneously on different portions of the data.

Parallel Suffix is just the reversed version of prefix:

- Also: elements needs to be **processed in parallel**
- This time each element is **combined with its <u>successors</u>**

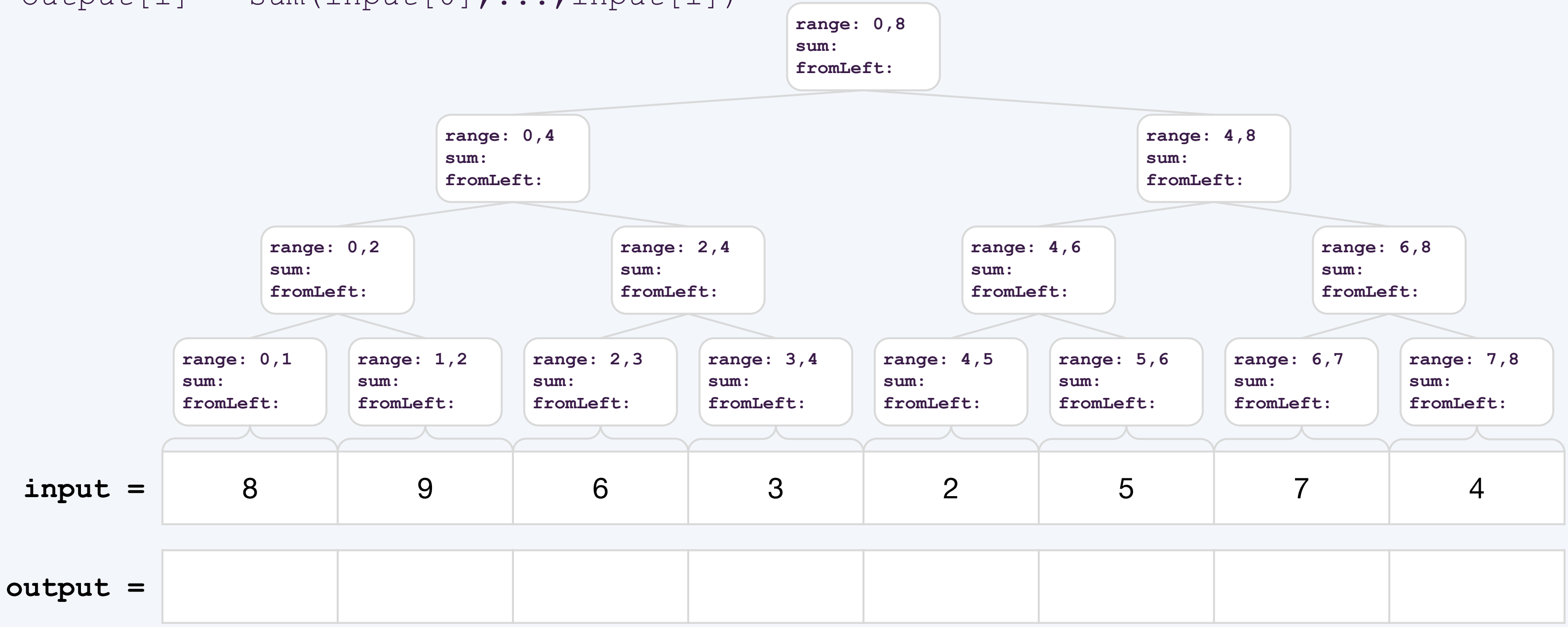
Problem 0

Parallel Prefix Sum
Parallel Suffix Sum

Problem 0a

1. Divide problem into parallel pieces
   `cutoff = 1`

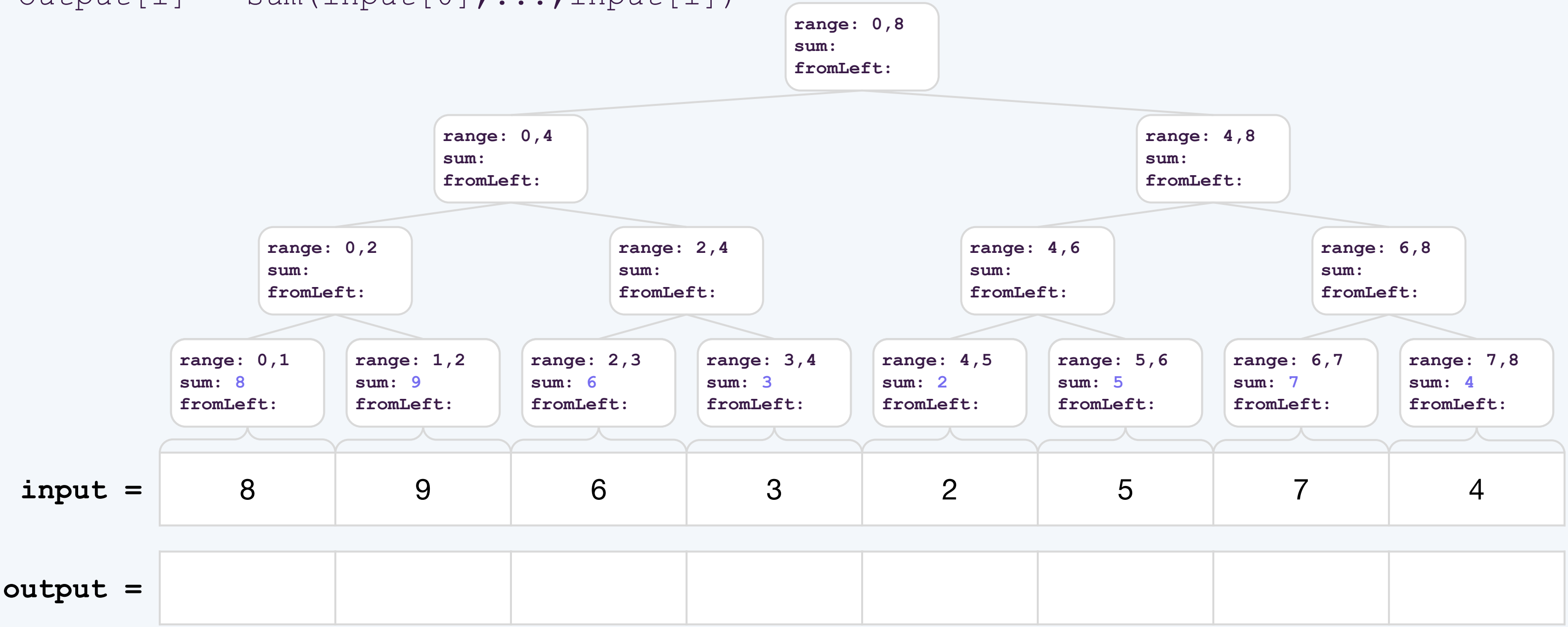
Given an array `input`, output an array such that
`output[i] = sum(input[0],...,input[i])`



Problem 0a

1. Divide problem into parallel pieces
   `cutoff = 1`
2. Find `sum` at cutoff
   `leaves[i].sum = input[i]`

Given an array `input`, output an array such that
`output[i] = sum(input[0],...,input[i])`

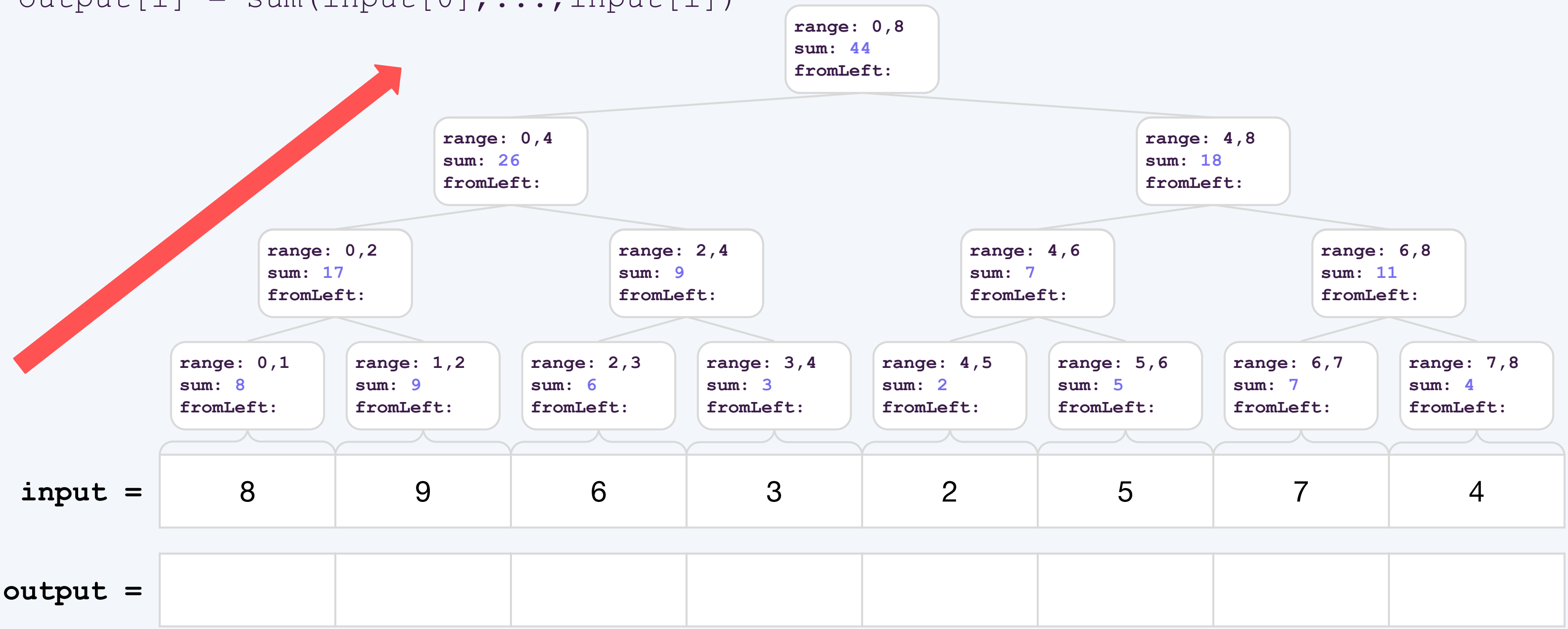


Problem 0a

1. Divide problem into parallel pieces
   `cutoff = 1`
2. Find `sum` at cutoff
   `leaves[i].sum = input[i]`
3. Propagate `sum` up
   `parent.sum = left.sum + right.sum`

Given an array `input`, output an array such that

`output[i] = sum(input[0],...,input[i])`

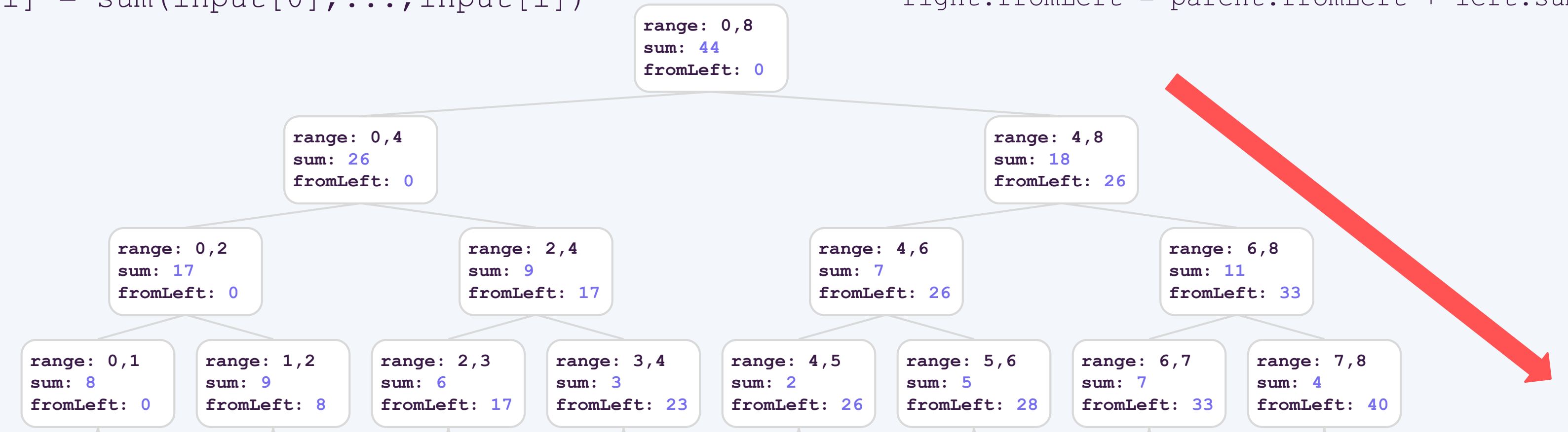


Problem 0a

1. Divide problem into parallel pieces
   `cutoff = 1`
2. Find `sum` at cutoff
   `leaves[i].sum = input[i]`
3. Propagate `sum` up
   `parent.sum = left.sum + right.sum`
4. Propagate `fromLeft` down
   `left.fromLeft = parent.fromLeft`
   `right.fromLeft = parent.fromLeft + left.sum`

Given an array `input`, output an array such that

`output[i] = sum(input[0],...,input[i])`



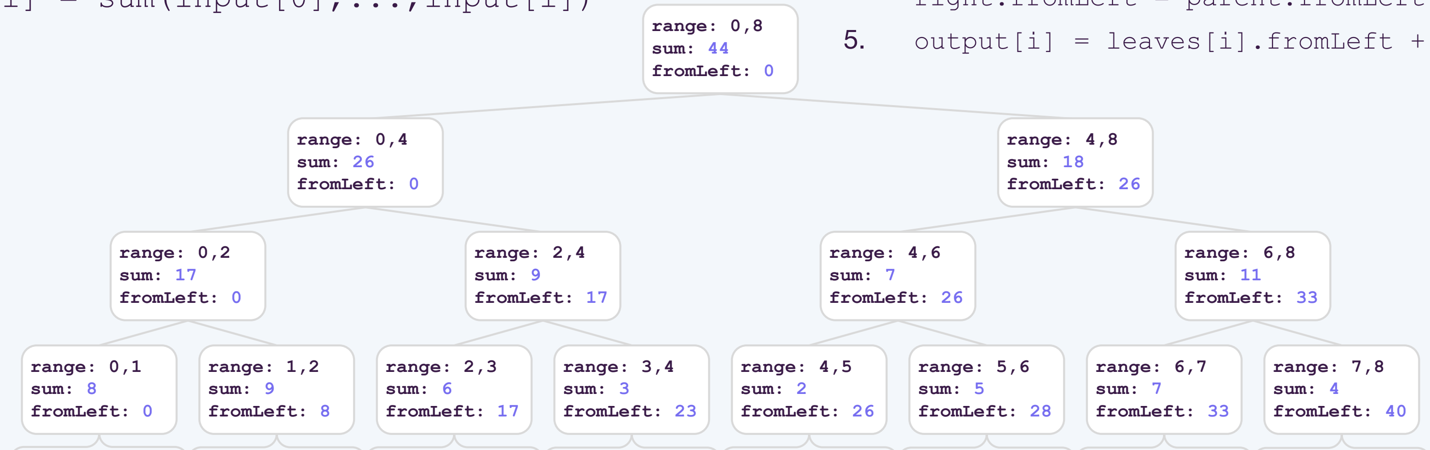
input =	8	9	6	3	2	5	7	4
output =								

Problem 0a

Given an array `input`, output an array such that

`output[i] = sum(input[0],...,input[i])`

1. Divide problem into parallel pieces
   `cutoff = 1`
2. Find `sum` at cutoff
   `leaves[i].sum = input[i]`
3. Propagate `sum` up
   `parent.sum = left.sum + right.sum`
4. Propagate `fromLeft` down
   `left.fromLeft = parent.fromLeft`
   `right.fromLeft = parent.fromLeft + left.sum`
5. `output[i] = leaves[i].fromLeft + input[i]`



input =	8	9	6	3	2	5	7	4
output =	8	17	23	26	28	33	40	44

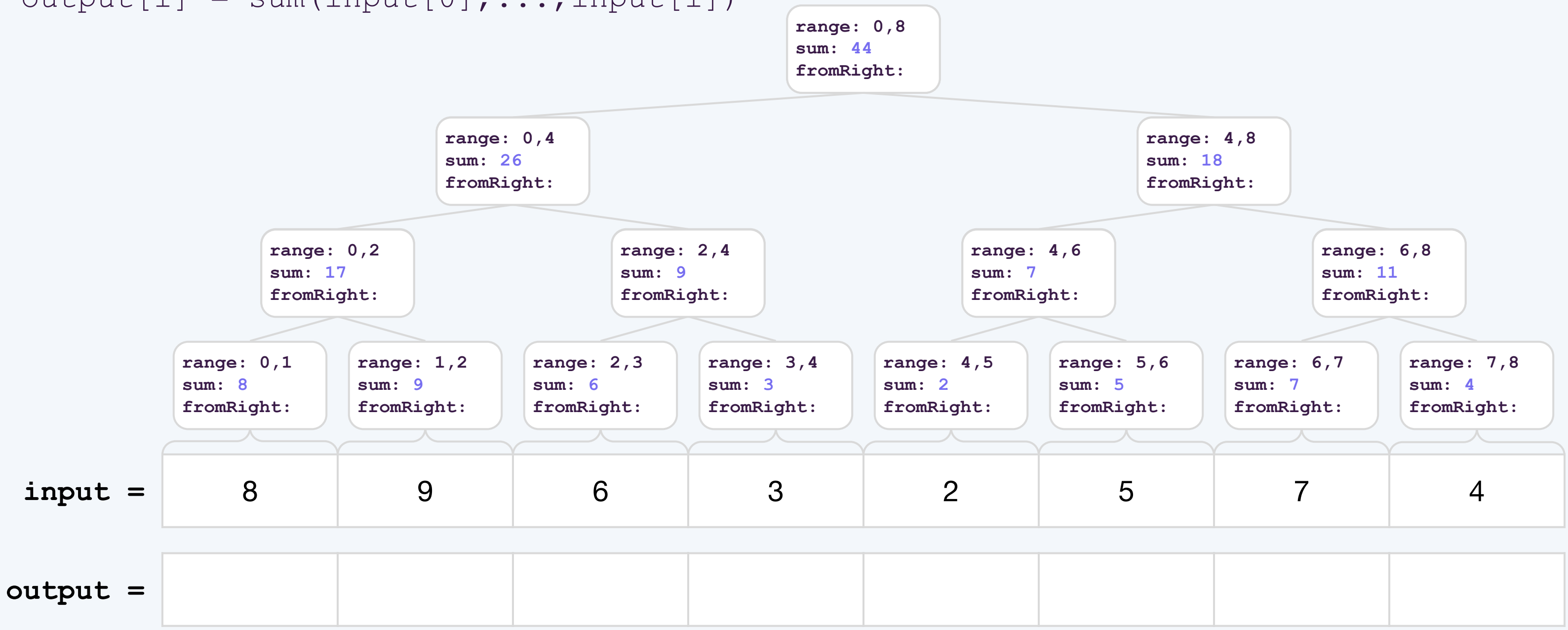
Problem 0b

Same steps to calculate **sum**

1. Divide problem into parallel pieces
   `cutoff = 1`
2. Find `sum` at cutoff
   `leaves[i].sum = input[i]`
3. Propagate `sum` up
   `parent.sum = left.sum + right.sum`

Given an array `input`, output an array such that

`output[i] = sum(input[0],...,input[i])`



Problem 0b

Same steps to calculate **sum**

1. Divide problem into parallel pieces

   `cutoff = 1`

2. Find `sum` at cutoff

   `leaves[i].sum = input[i]`

3. Propagate `sum` up

   `parent.sum = left.sum + right.sum`

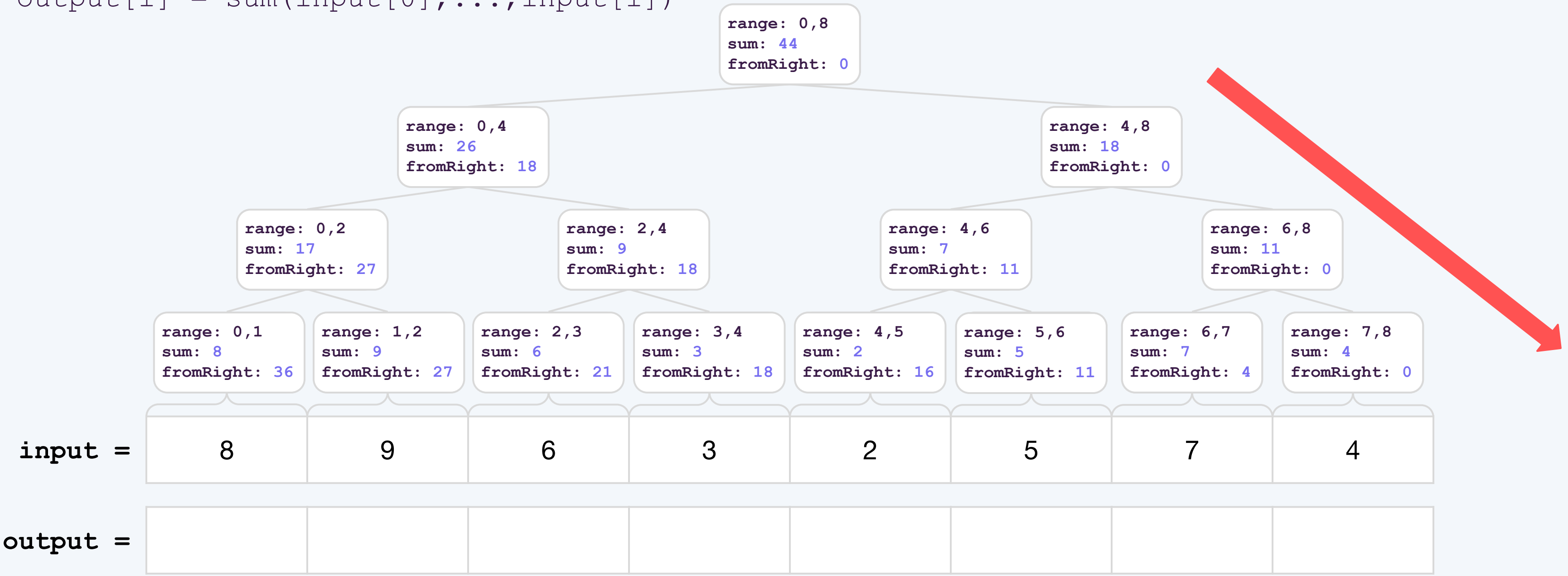
4. Propagate `fromRight` down

   `right.fromRight = parent.fromRight`

   `left.fromRight = parent.fromRight + right.sum`

Given an array `input`, output an array such that

`output[i] = sum(input[0],...,input[i])`



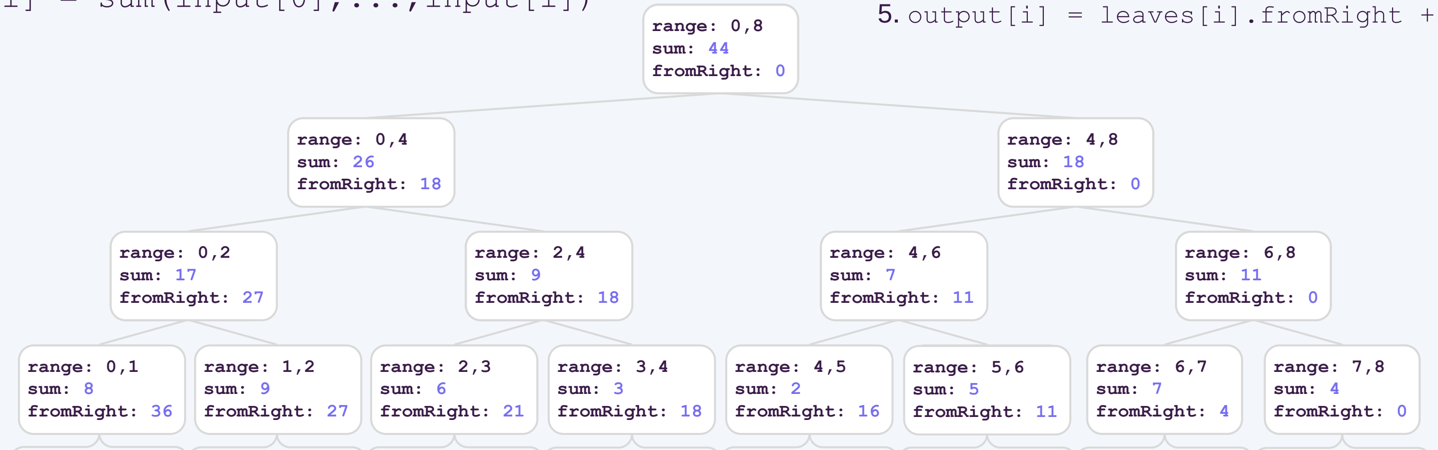
Problem 0b

Same steps to calculate **sum**

1. Divide problem into parallel pieces
   `cutoff = 1`
2. Find `sum` at cutoff
   `leaves[i].sum = input[i]`
3. Propagate `sum` up
   `parent.sum = left.sum + right.sum`
4. Propagate `fromRight` down
   `right.fromRight = parent.fromRight`
   `left.fromRight = parent.fromRight + right.sum`
5. `output[i] = leaves[i].fromRight + input[i]`

Given an array `input`, output an array such that

`output[i] = sum(input[0],...,input[i])`



input =	8	9	6	3	2	5	7	4
output =	44	36	27	21	18	16	11	4

Problem 1

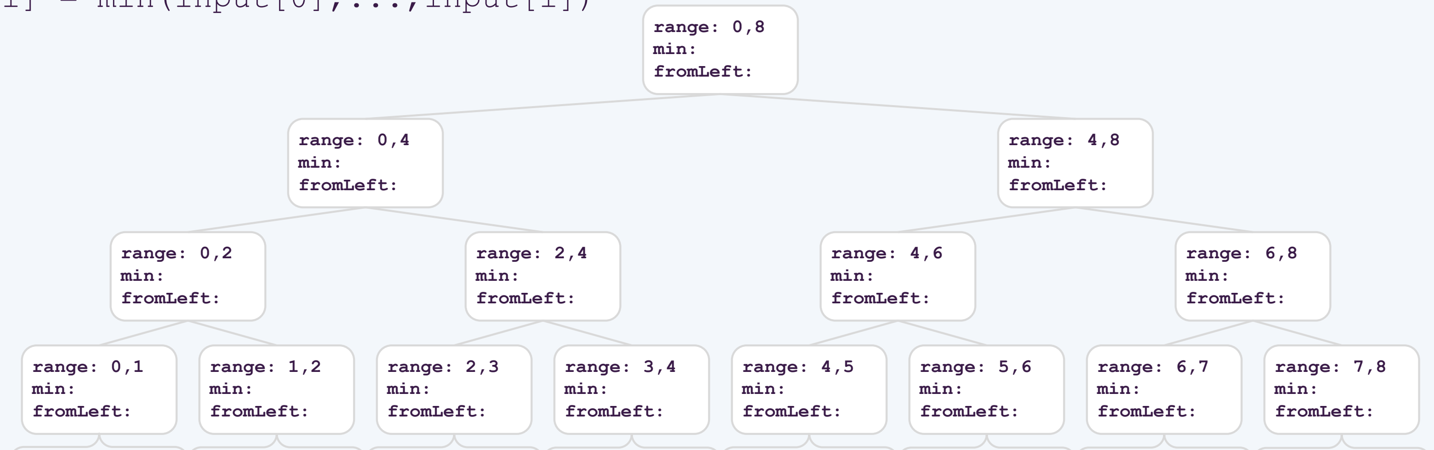
Parallel Prefix FindMin

Problem 1

1. Divide problem into parallel pieces
   `cutoff = 1`

Given an array `input`, output an array such that

`output[i] = min(input[0],...,input[i])`



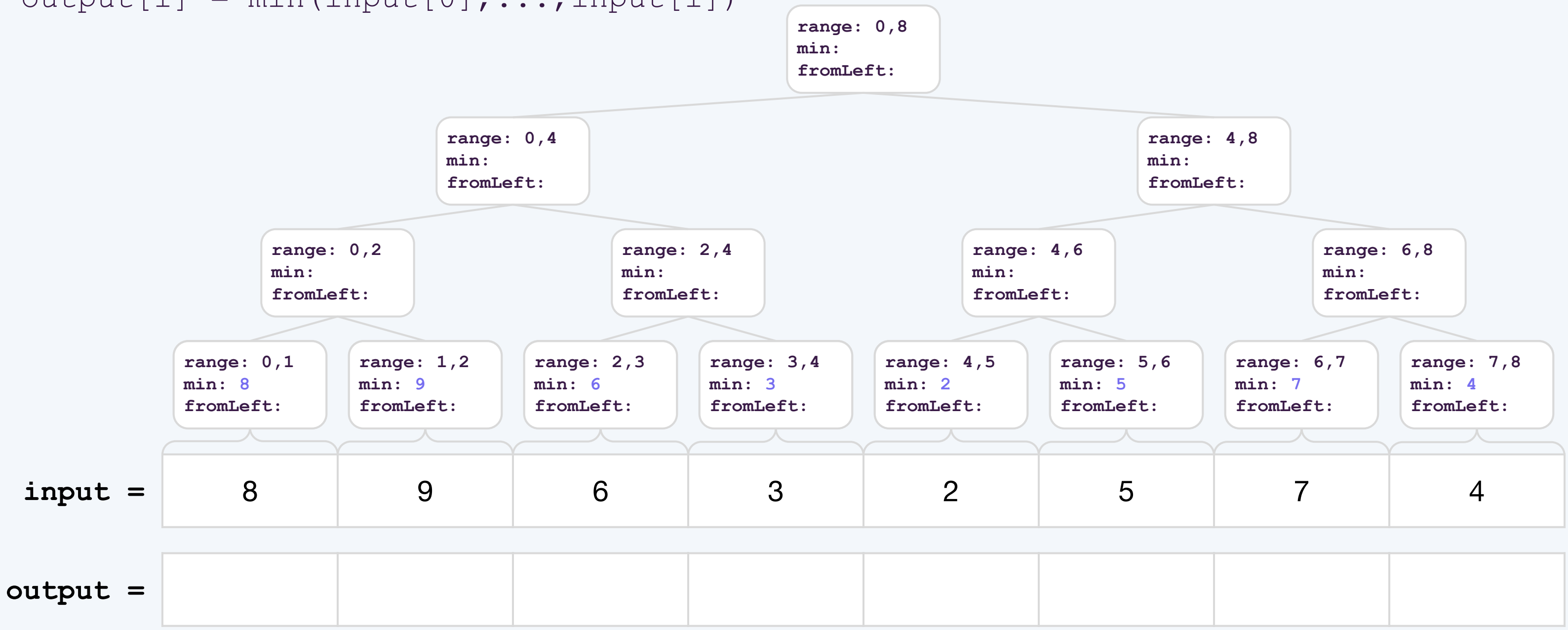
input =	8	9	6	3	2	5	7	4
output =								

Problem 1

1. Divide problem into parallel pieces
   `cutoff = 1`
2. Find `min` at cutoff
   `leaves[i].min = input[i]`

Given an array `input`, output an array such that

`output[i] = min(input[0],...,input[i])`

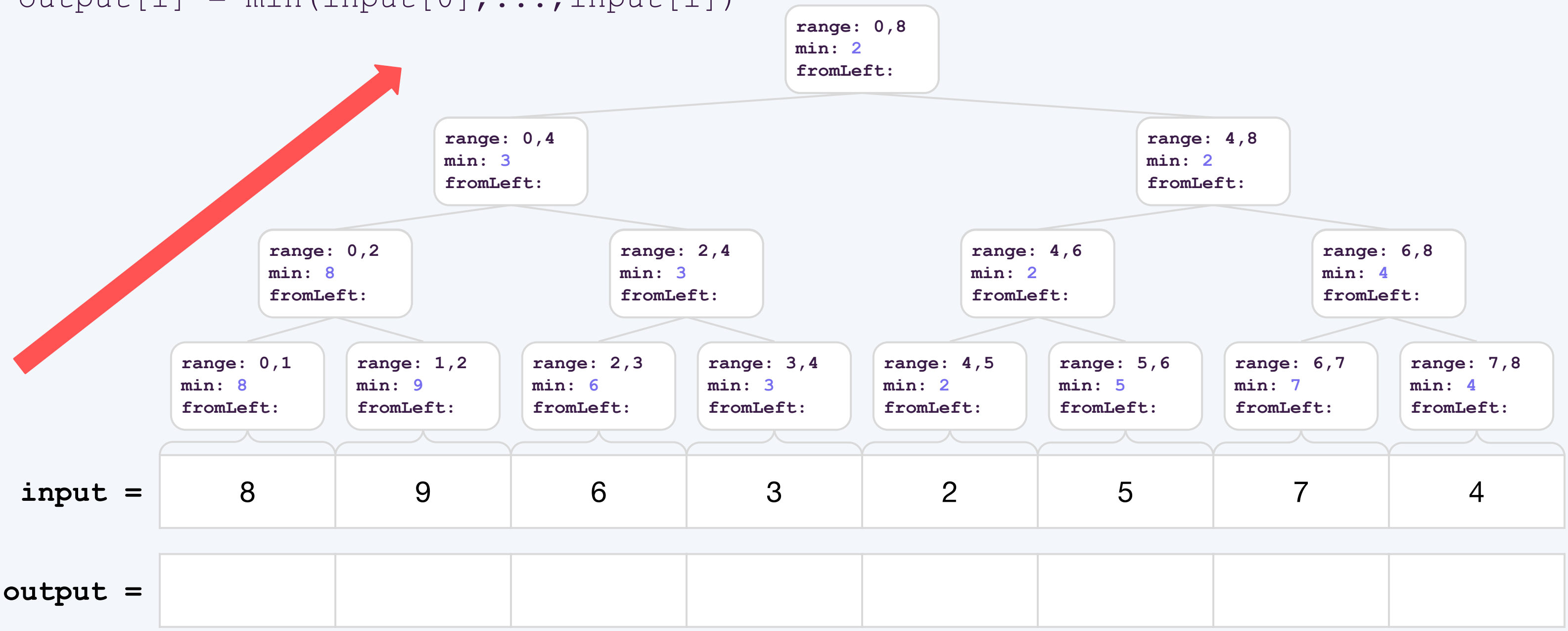


Problem 1

1. Divide problem into parallel pieces
   `cutoff = 1`
2. Find `min` at cutoff
   `leaves[i].min = input[i]`
3. Propagate `min` up
   `parent.min = min(left.min, right.min)`

Given an array `input`, output an array such that

`output[i] = min(input[0],...,input[i])`

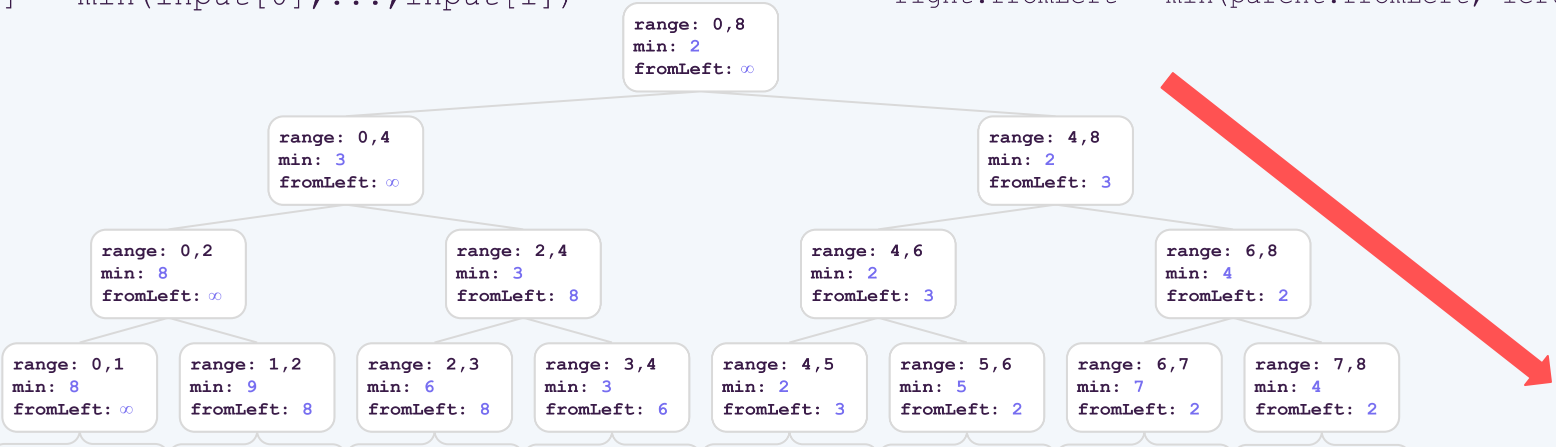


Problem 1

Given an array `input`, output an array such that

`output[i] = min(input[0],...,input[i])`

1. Divide problem into parallel pieces
   `cutoff = 1`
2. Find `min` at cutoff
   `leaves[i].min = input[i]`
3. Propagate `min` up
   `parent.min = min(left.min, right.min)`
4. Propagate `fromLeft` down
   `left.fromLeft = parent.fromLeft`
   `right.fromLeft = min(parent.fromLeft, left.min)`



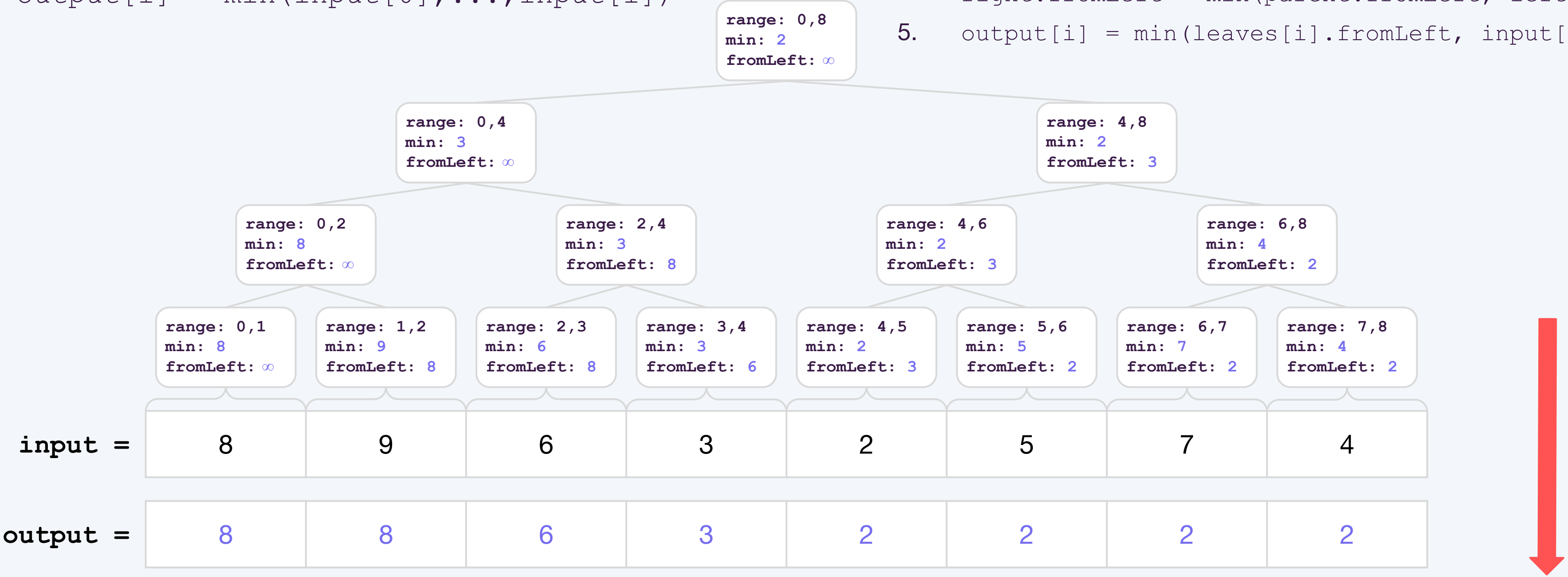
input =	8	9	6	3	2	5	7	4
output =								

Problem 1

Given an array `input`, output an array such that

`output[i] = min(input[0],...,input[i])`

1. Divide problem into parallel pieces
   `cutoff = 1`
2. Find `min` at cutoff
   `leaves[i].min = input[i]`
3. Propagate `min` up
   `parent.min = min(left.min, right.min)`
4. Propagate `fromLeft` down
   `left.fromLeft = parent.fromLeft`
   `right.fromLeft = min(parent.fromLeft, left.min)`
5. `output[i] = min(leaves[i].fromLeft, input[i])`



Problem 2

Parallel Pack

Problem 2 Overview

Given an array `input`, output an array that contains only the elements that are less than 10

1. **Parallel map** to compute `bits` array where `bits[i] = (input[i] < 10)`

input =	12	5	-8	34	6	10	2	7
bits =	0	1	1	0	1	0	1	1

2. **Parallel prefix sum** on `bits` array to compute `bitsum` array

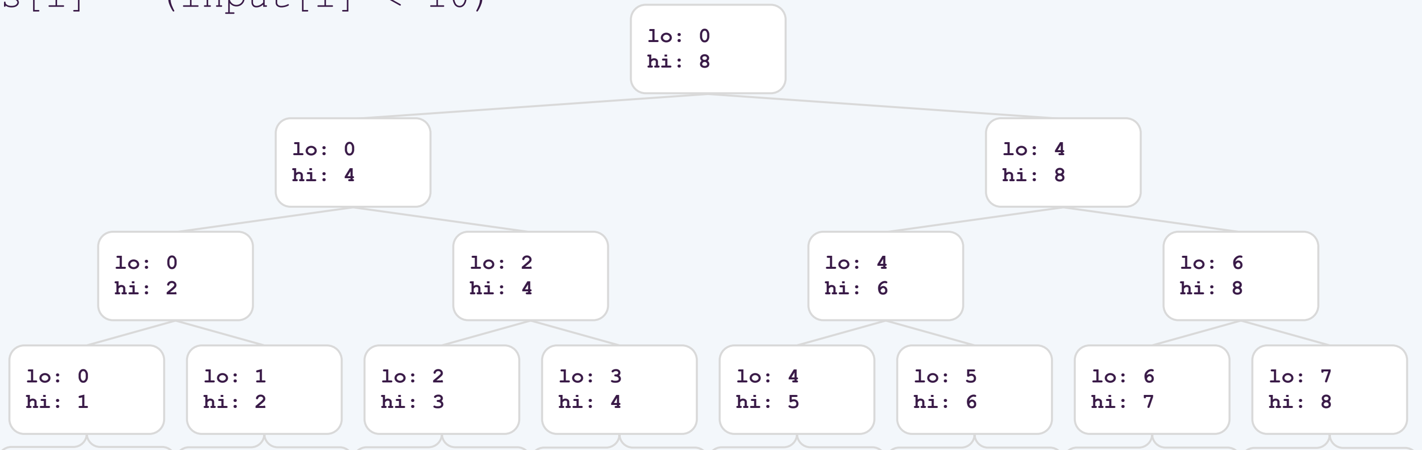
bitsum =	0	1	2	2	3	3	4	5

3. **Parallel map** to produce `output` array where `output[bitsum[i] - 1] = input[i]` if `bits[i] == 1`

output =	5	-8	6	2	7

Problem 2

1. **Parallel map** to compute `bits` array where
   `bits[i] = (input[i] < 10)`

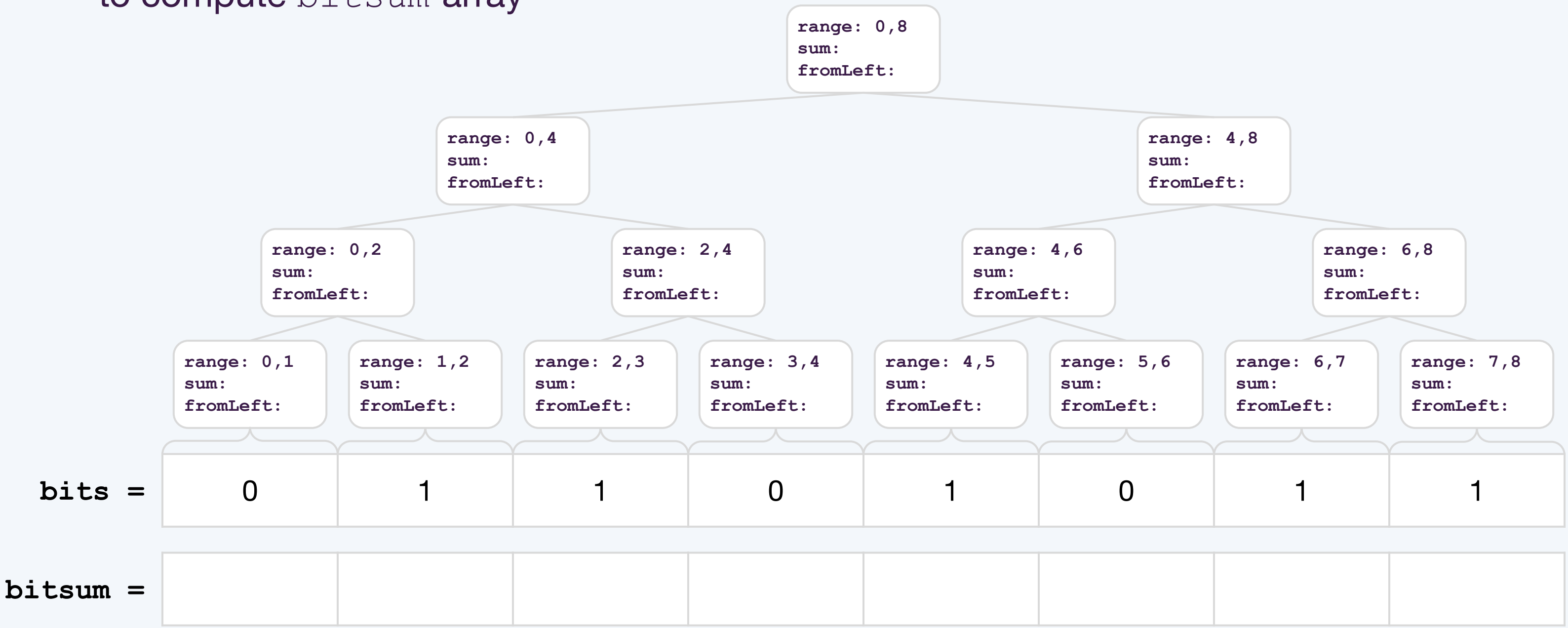


input =	12	5	-8	34	6	10	2	7
bits =	0	1	1	0	1	0	1	1

Problem 2

1. Divide problem into parallel pieces
   `cutoff = 1`

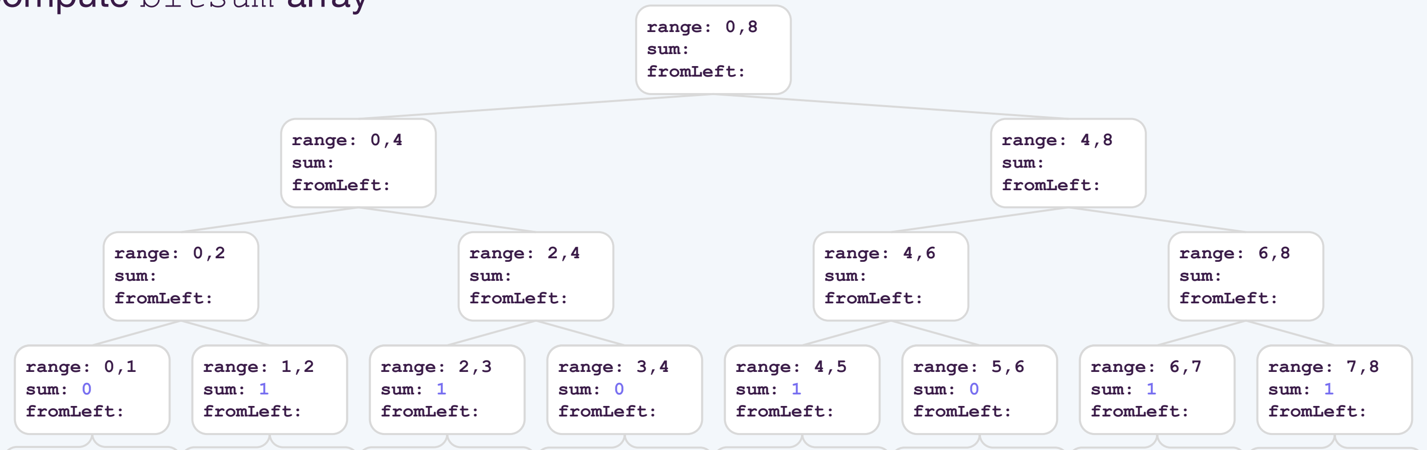
2. **Parallel prefix sum** on `bits` array to compute `bitsum` array



Problem 2

1. Divide problem into parallel pieces
   `cutoff = 1`
2. Find `sum` at cutoff
   `leaves[i].sum = bits[i]`

2. **Parallel prefix sum** on `bits` array to compute `bitsum` array

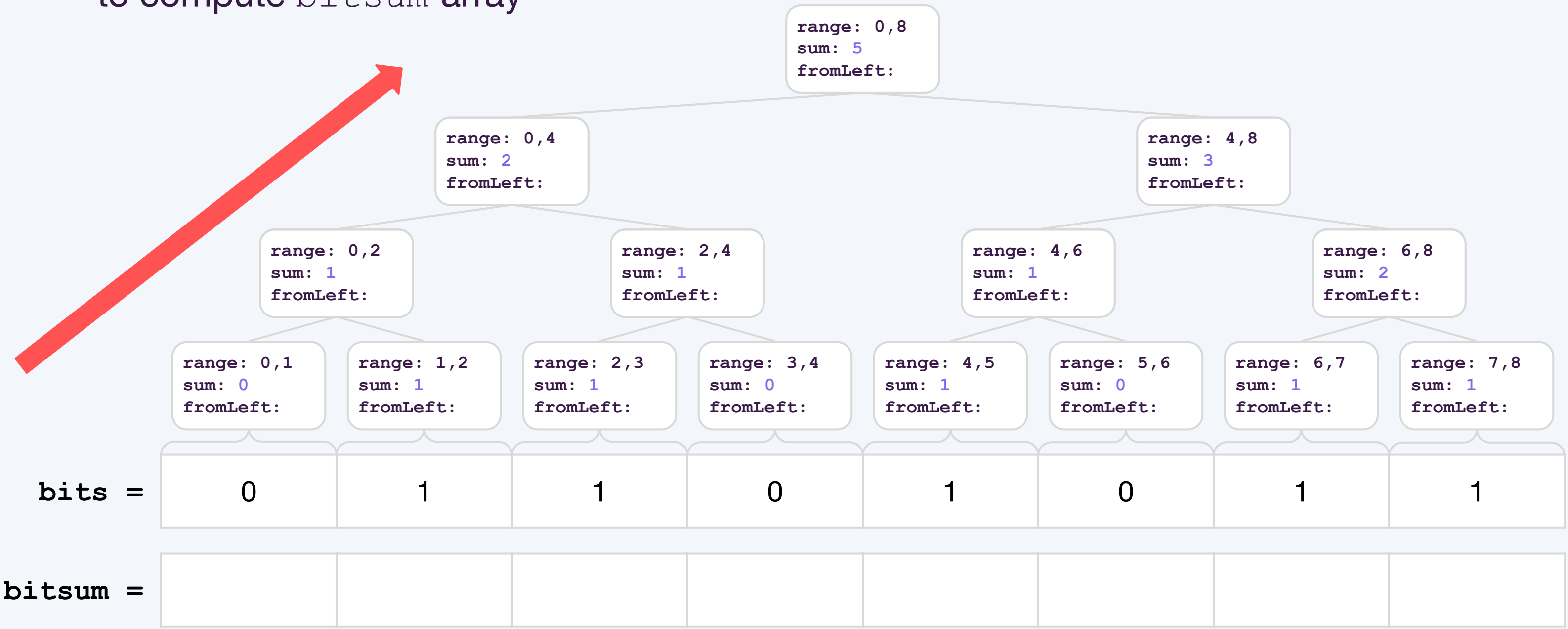


bits =	0	1	1	0	1	0	1	1
bitsum =								

Problem 2

1. Divide problem into parallel pieces
   `cutoff = 1`
2. Find `sum` at cutoff
   `leaves[i].sum = bits[i]`
3. Propagate `sum` up
   `parent.sum = left.sum + right.sum`

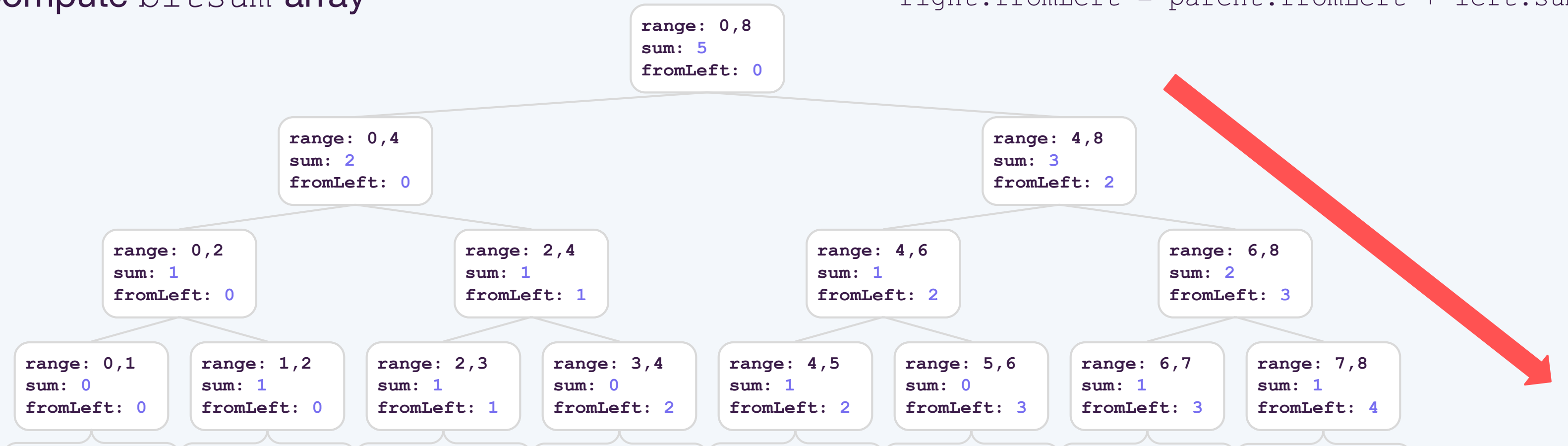
2. **Parallel prefix sum** on `bits` array to compute `bitsum` array



Problem 2

2. **Parallel prefix sum** on `bits` array to compute `bitsum` array

1. Divide problem into parallel pieces
   `cutoff = 1`
2. Find `sum` at cutoff
   `leaves[i].sum = bits[i]`
3. Propagate `sum` up
   `parent.sum = left.sum + right.sum`
4. Propagate `fromLeft` down
   `left.fromLeft = parent.fromLeft`
   `right.fromLeft = parent.fromLeft + left.sum`

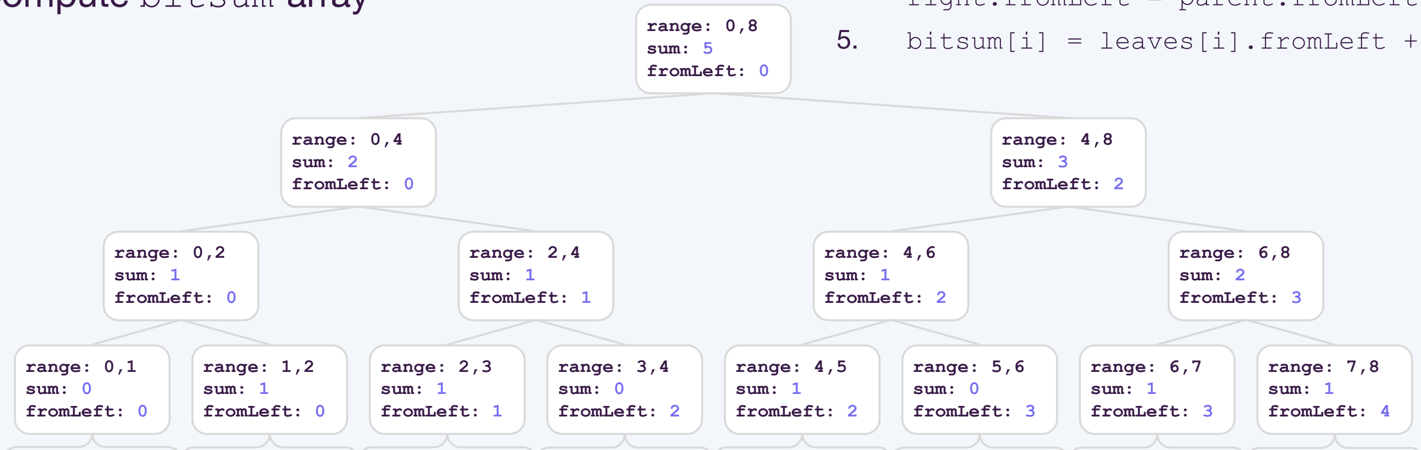


bits =	0	1	1	0	1	0	1	1
bitsum =								

Problem 2

2. **Parallel prefix sum** on `bits` array to compute `bitsum` array

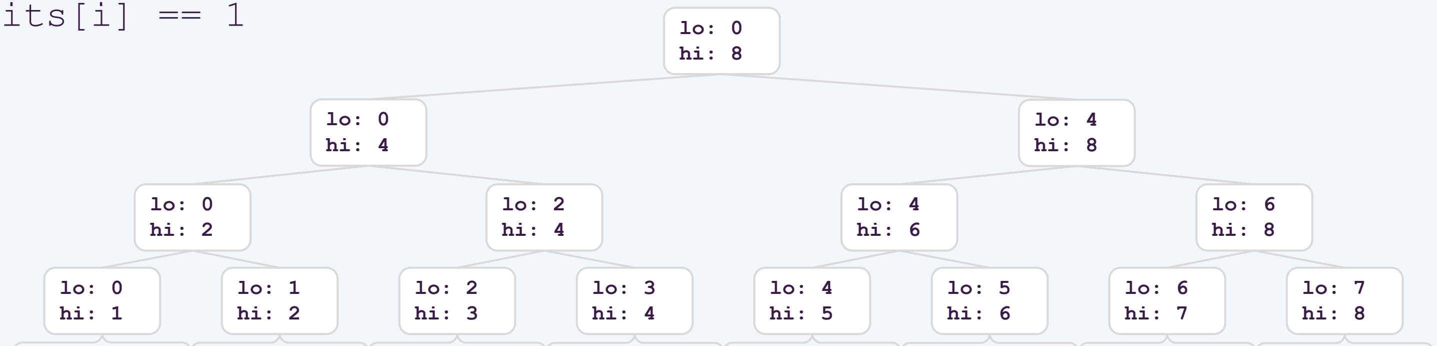
1. Divide problem into parallel pieces
   `cutoff = 1`
2. Find `sum` at cutoff
   `leaves[i].sum = bits[i]`
3. Propagate `sum` up
   `parent.sum = left.sum + right.sum`
4. Propagate `fromLeft` down
   `left.fromLeft = parent.fromLeft`
   `right.fromLeft = parent.fromLeft + left.sum`
5. `bitsum[i] = leaves[i].fromLeft + bits[i]`



bits =	0	1	1	0	1	0	1	1
bitsum =	0	1	2	2	3	3	4	5

Problem 2

3. **Parallel map** to produce `output` array where `output[bitsum[i] - 1] = input[i]` if `bits[i] == 1`



input =	12	5	-8	34	6	10	2	7
bits =	0	1	1	0	1	0	1	1
bitsum =	0	1	2	2	3	3	4	5

output =	5	-8	6	2	7

Problem 3

Problem 3

a) **Define work and span.**
Work is the runtime of a program on **one** processor.
Span is the runtime of a program on **infinitely** many processors.

b) **How do we calculate work and span?**
Work is the sum of all the work done by all processors
Span is the sum of all the work done by the longest dependence chain / parallel ‘tree’ branch

c) **Does adding more processors affect the work or span?**
Neither. Work is defined for one processor and span is defined for infinitely many processors.
Adding more processors will not affect either work or span.

Problem 3

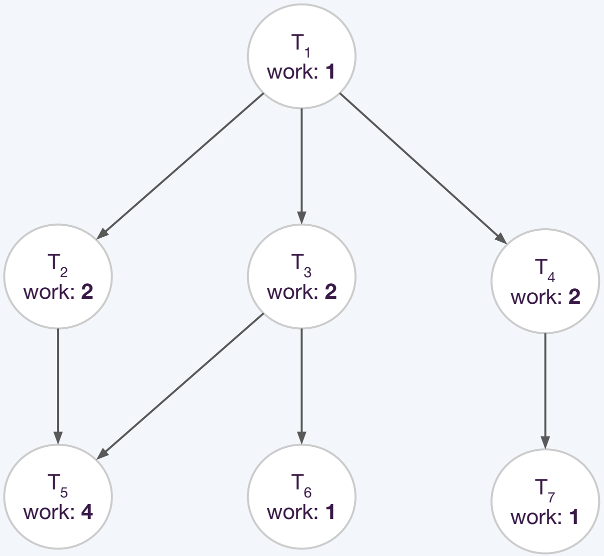
d) What is the work and span of the graph?

Work = 1 + 2 + 2 + 2 + 4 + 1 + 1

= 13

Span $(T_1 \rightarrow T_2 \rightarrow T_5)$ = 1 + 2 + 4

= 7



Amdahl's Law Review

- Amdahl's Law describes the potential speedup of a task using multiple processors compared to a single processor.

- Speedup with P processors is defined as $\frac{T_1}{T_p} \leq \frac{1}{S + \frac{1-S}{P}}$
- Where S is the proportion of the task that requires to be done sequentially

Problem 4

e) Suppose a program needs to do 20% of its work sequentially (the rest can be parallelized), what is the maximum speed up with 4 processors?

Recall Amdahl's Law:

Amdahl's Law

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

Problem 4

Recall Amdahl's Law:

Amdahl's Law

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

e) Suppose a program needs to do 20% of its work sequentially (the rest can be parallelized), what is the maximum speed up with 4 processors?

$$S = \frac{1}{0.2 + \frac{1-0.2}{4}} = 2.5$$

Thank You!