Section 8: Parallel Prefix

0a. Parallel Prefix Sum

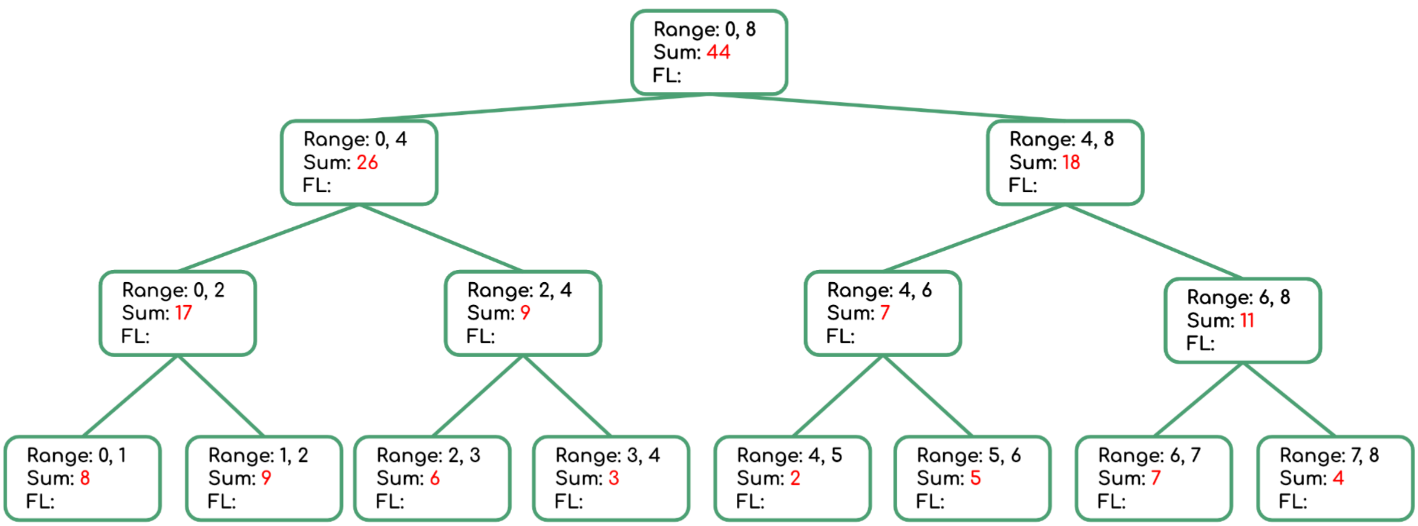
Given input array `[8, 9, 6, 3, 2, 5, 7, 4]`, output an array such that each `output[i] = sum(array[0], array[1], ..., array[i])`.

Use the [Parallel Prefix Sum] algorithm from lecture. Show the intermediate steps. Draw the input and output arrays, and for each step, show the tree of the recursive task objects that would be created (where a node's child is for two problems of half the size) and the fields each node needs. Do not use a sequential cut-off.

First pass: fill out the *sum* field starting from leaf nodes to the top by starting with each leaf node's value as its *sum*, then combining parallel subproblems by taking the sum of each side. This can be calculated with the following expressions:

leaves[i].sum = input[i]

p.sum = p.left.sum + p.right.sum



Input	8	9	6	3	2	5	7	4
Output								

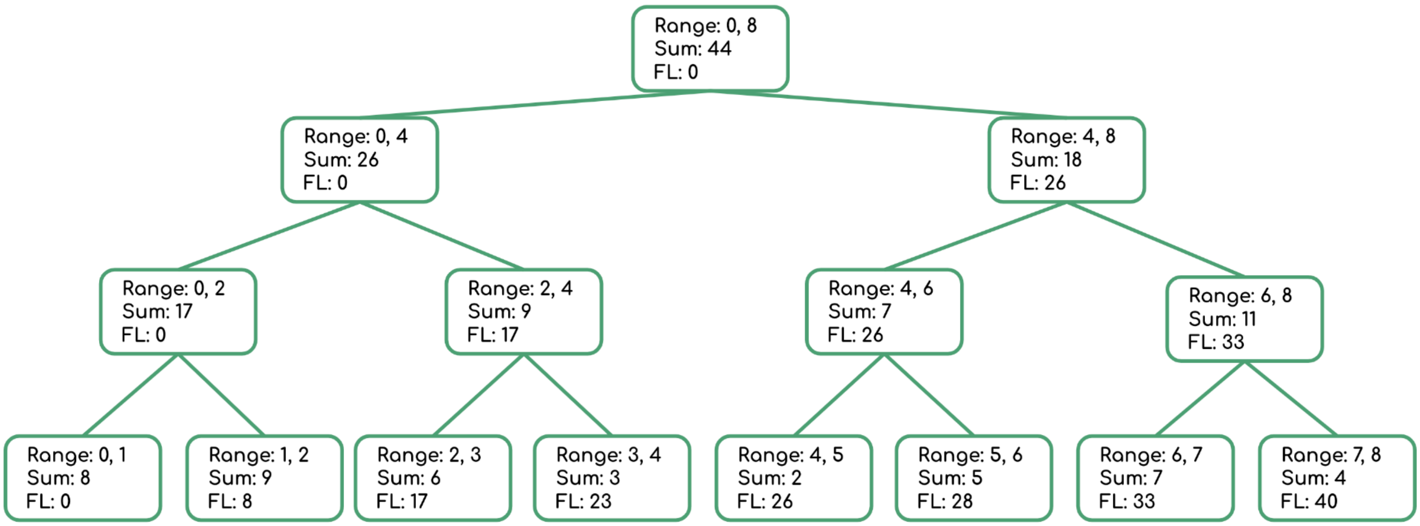
Second pass: fill out the *FL* (“from left”) field starting from the top down to the leaf nodes to represent the sum of the *prefix* of this subproblem’s range, that is, the sum of everything to the *left* of this node. This can calculated with the following expressions:

p.right.FL = p.FL + p.left.sum

p.left.FL = p.FL

Then fill the output array with the *sum* and *FL* fields at the leaf node level:

output[i] = leaves[i].FL + input[i]



Input	8	9	6	3	2	5	7	4
Output	8	17	23	26	28	33	40	44

0b. Parallel Suffix Sum

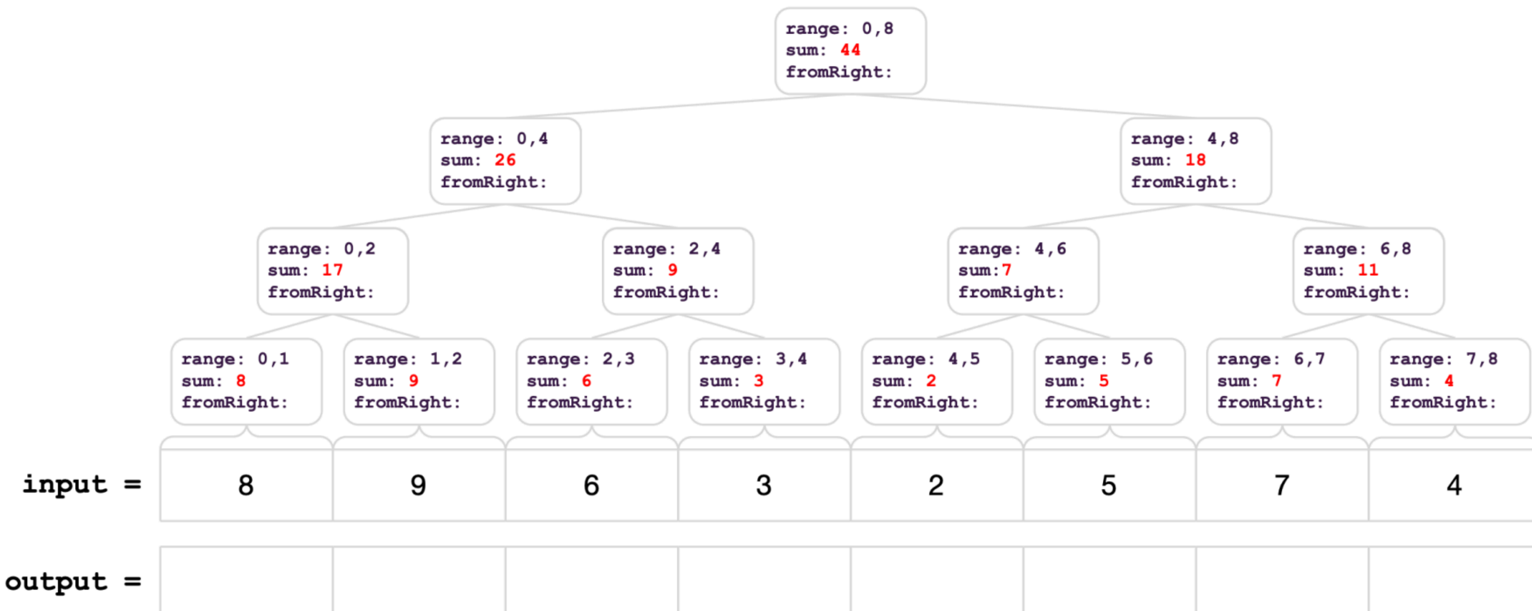
Given input array `[8, 9, 6, 3, 2, 5, 7, 4]`, output an array such that each `output[i] = sum(array[0], array[1], ..., array[i])`.

Use the Parallel Suffix Sum algorithm. Show the intermediate steps. Draw the input and output arrays, and for each step, show the tree of the recursive task objects that would be created (where a node's child is for two problems of half the size) and the fields each node needs. Do not use a sequential cut-off.

First pass: fill out the *sum* field starting from leaf nodes to the top by starting with each leaf node's value as its *sum*, then combining parallel subproblems by taking the sum of each side. This can be calculated with the following expressions:

leaves[i].sum = input[i]

p.sum = p.left.sum + p.right.sum



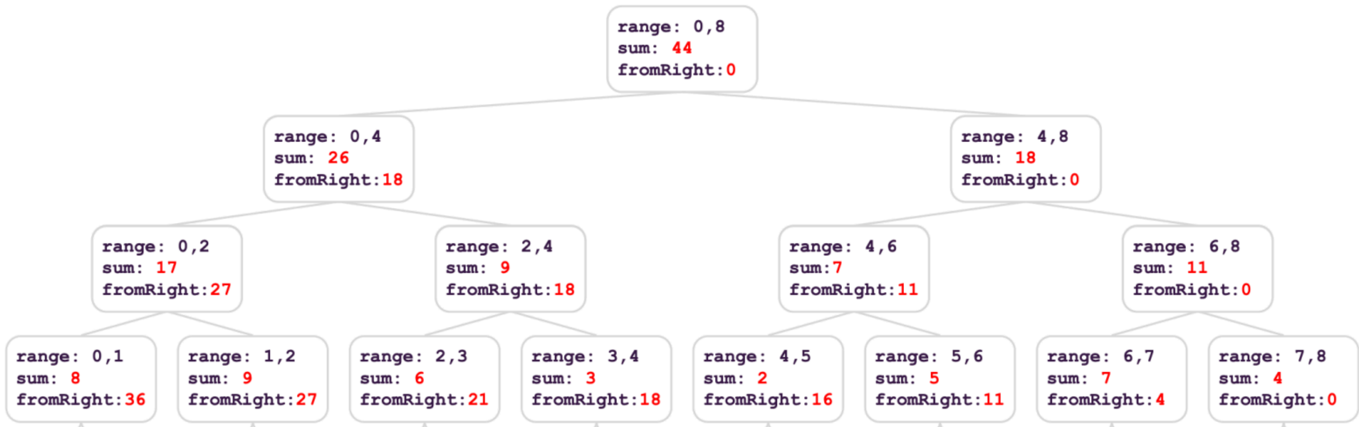
Second pass: fill out the *fromRight* field starting from the top down to the leaf nodes to represent the sum of the *prefix* of this subproblem's range, that is, the sum of everything to the *right* of this node. This can calculated with the following expressions:

p.left.fromRight = p.fromRight + p.right.sum

p.right.fromRight = p.fromRight

Then fill the output array with the *sum* and fromRight fields at the leaf node level:

output[i] = leaves[i].fromRight + input[i]



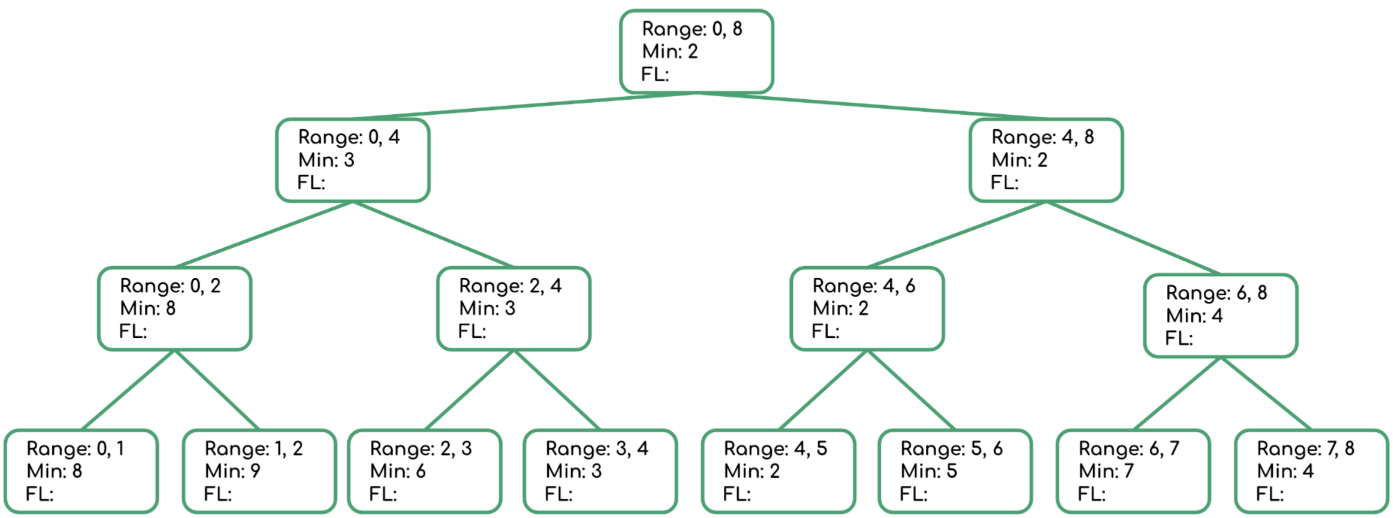
input =	8	9	6	3	2	5	7	4
output =	44	36	27	21	18	16	11	4

1. Parallel Prefix FindMin

Given input array `[8, 9, 6, 3, 2, 5, 7, 4]`, output an array such that each `output[i] = min(array[0], array[1], ..., array[i])`. Show all steps, as above.

First pass: fill out the *min* field starting from leaf nodes to the top by starting with each leaf node's value as its *min*, then combining parallel subproblems by taking the min of each side. This can be calculated with the following expressions:

leaves[i].min = input[i]
p.min = min(p.left.min, p.right.min)



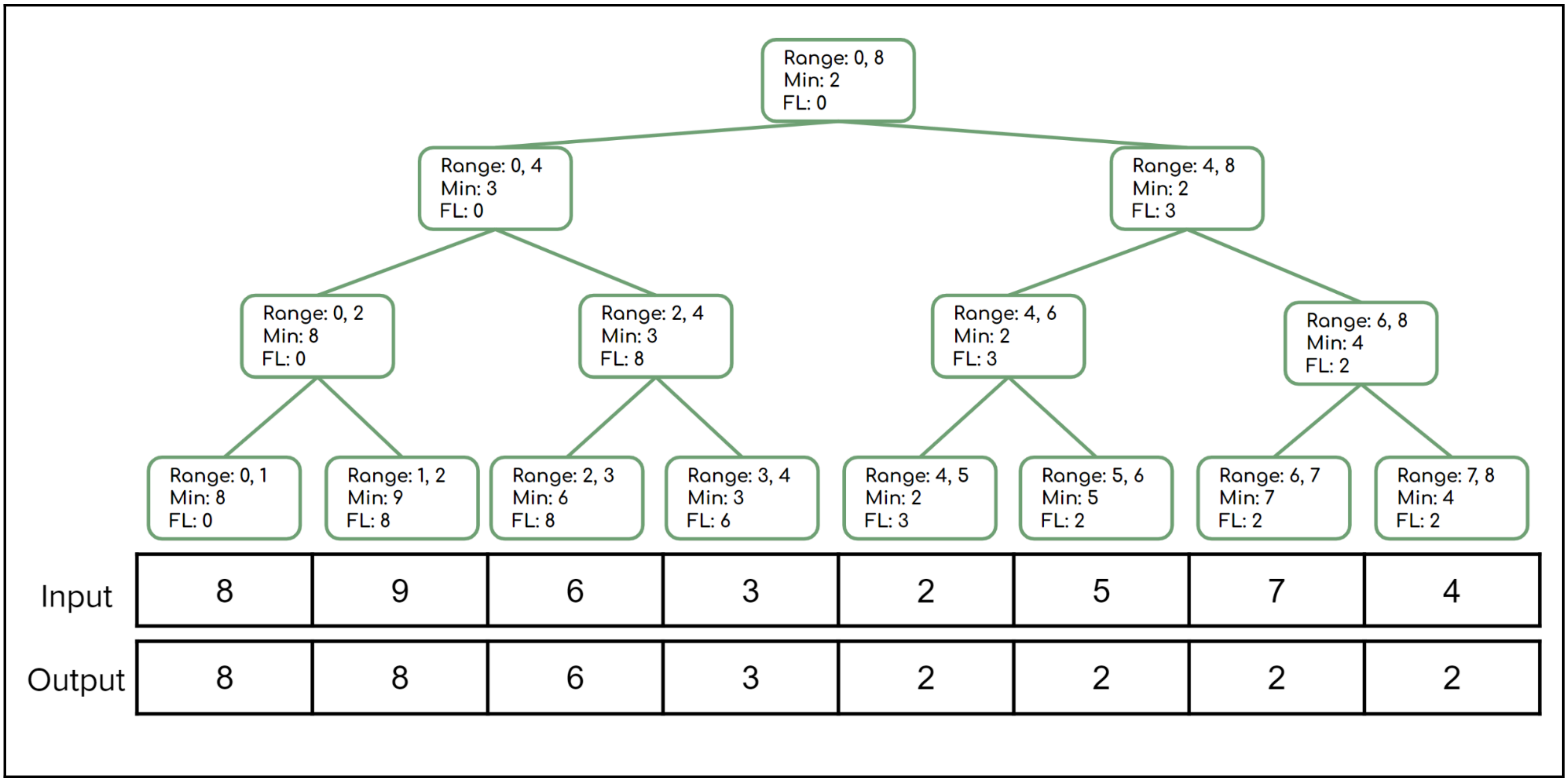
Input	8	9	6	3	2	5	7	4
Output								

Second pass: fill out the *FL* ("from left") field starting from the top down to the leaf nodes to represent the minimum value of the *prefix* of this subproblem's range, that is, the min of everything to the *left* of this node. This can calculated with the following expressions:

p.right.FL = min(p.FL, p.left.min)
p.left.FL = p.FL

Then fill the output array with the *min* and *FL* fields at the leaf node level:

output[i] = min(leaves[i].FL, input[i])



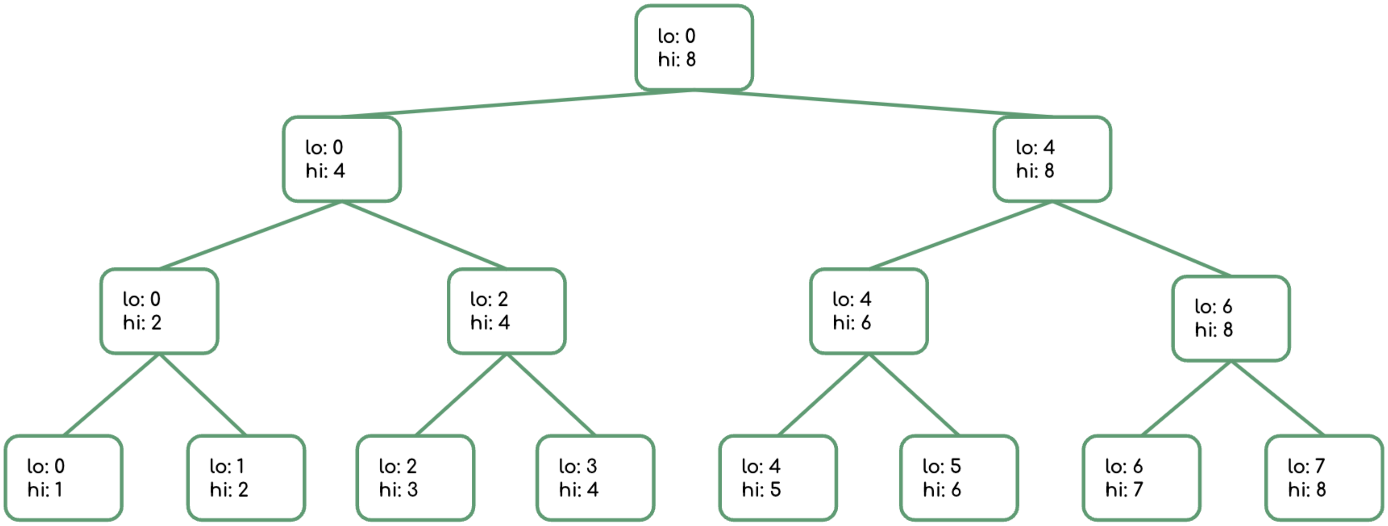
Range: 0, 8
Min: 2
FL: 0
Range: 0, 4
Min: 3
FL: 0
Range: 4, 8
Min: 2
FL: 3
Range: 0, 2
Min: 8
FL: 0
Range: 2, 4
Min: 3
FL: 8
Range: 4, 6
Min: 2
FL: 3
Range: 6, 8
Min: 4
FL: 2
Range: 0, 1
Min: 8
FL: 0
Range: 1, 2
Min: 9
FL: 8
Range: 2, 3
Min: 6
FL: 8
Range: 3, 4
Min: 3
FL: 6
Range: 4, 5
Min: 2
FL: 3
Range: 5, 6
Min: 5
FL: 2
Range: 6, 7
Min: 7
FL: 2
Range: 7, 8
Min: 4
FL: 2
Input
8 9 6 3 2 5 7 4
Output
8 8 6 3 2 2 2 2

2. Parallel Pack

Given input array `[12, 5, -8, 34, 6, 10, 2, 7]`, output an array that contains only the elements that are less than 10.

Use the Parallel Pack algorithm from lecture. Show the intermediate steps. Draw the input and output arrays, and for each step, show the tree of the recursive task objects that would be created (where a node's child is for two problems of half the size) and the fields each node needs. Do not use a sequential cut-off.

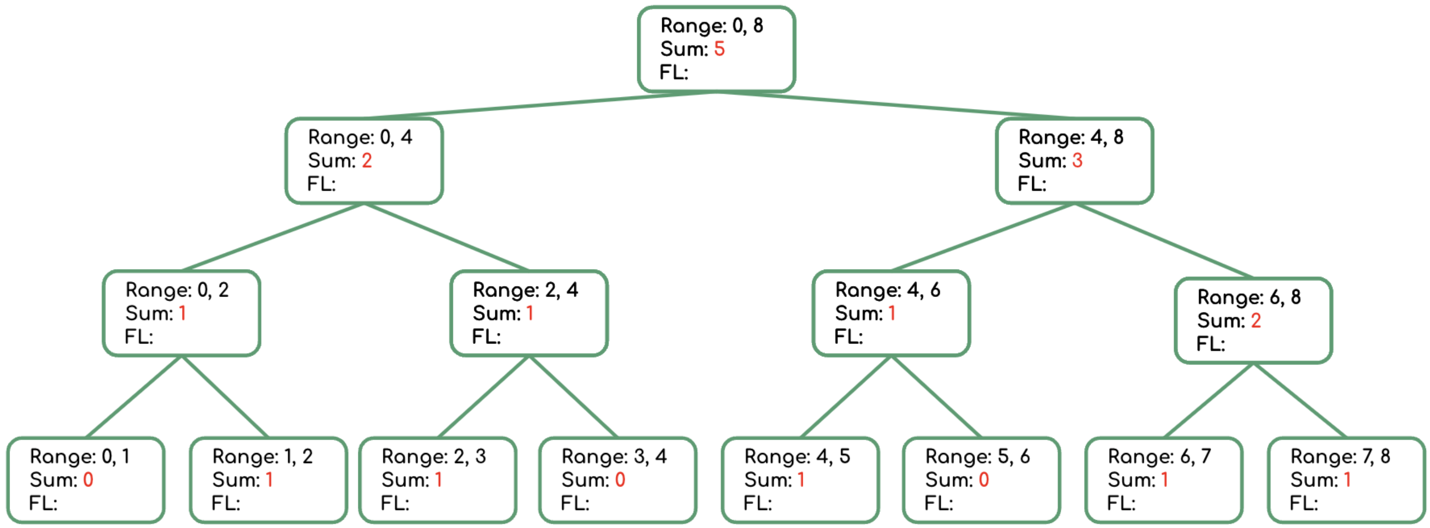
Step 1: parallel map to compute bits array such that bits[i] = 1 if input[i] < 10



Input	12	5	-8	34	6	10	2	7
Bits	0	1	1	0	1	0	1	1

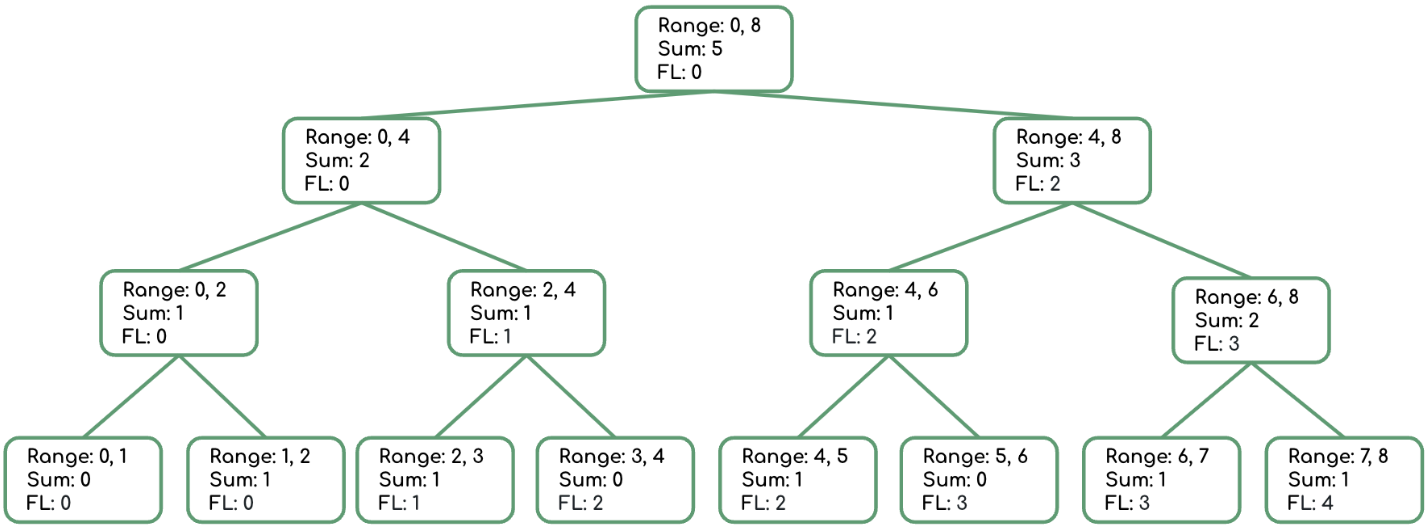
Step 2: parallel prefix sum to on bits array

- First pass: fill out the *sum* field starting from leaf nodes to the top by starting with each leaf node's value as its *sum*, then combining parallel subproblems by taking the sum of each side. This can be calculated with the following expressions:
  - leaves[i].sum = bits[i]
  - p.sum = p.left.sum + p.right.sum



Bits	0	1	1	0	1	0	1	1
Bitsum								

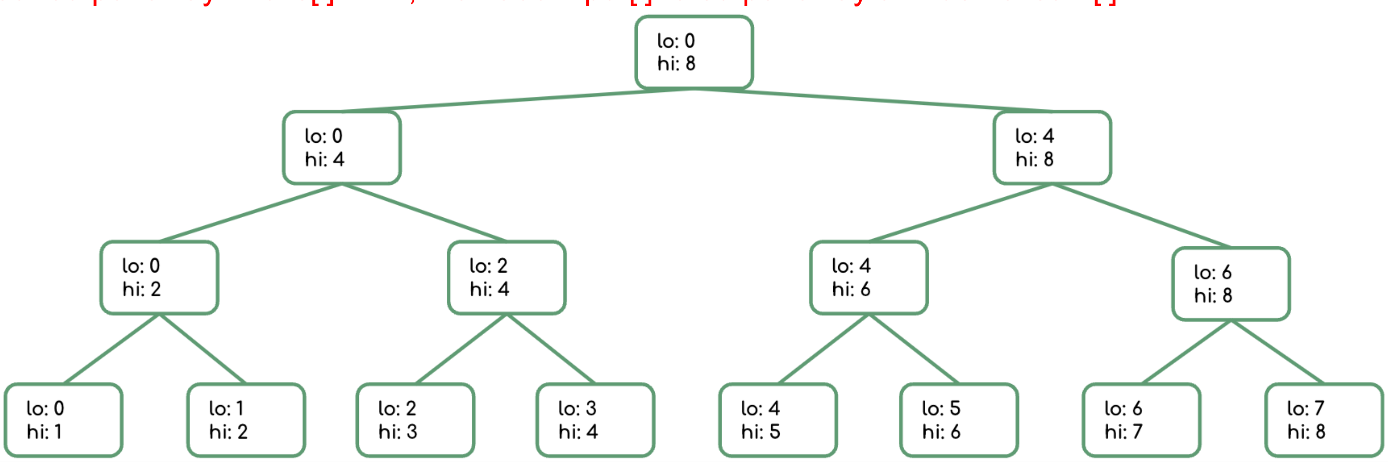
- Second pass: fill out the *FL* ("from left") field starting from the top down to the leaf nodes to represent the sum of the *prefix* of this subproblem's range, that is, the sum of everything to the *left* of this node. This can calculated with the following expressions:
    - p.right.FL = p.FL + p.left.sum
    - p.left.FL = p.FL
    - Then fill bitsum array with the *sum* and *FL* fields at the leaf node level:
        - bitsum[i] = leaves[i].FL + bits[i]



Bits	0	1	1	0	1	0	1	1
Bitsum	0	1	2	2	3	3	4	5

Step 3: parallel map to produce output array

- Create output array of size bitsum[n-1] where n is the size of input array.
- Fill out output array: if bits[i] == 1, then add input[i] to output array at index bitsum[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			

3. Work it Out [the Span]

a) Define work and span.

Work - how long the running time of a program would be with just one processor
Span - the running time with an infinite number of processors

b) How do we calculate work and span?

Work - sum all the work done by each processor
Span - calculate the longest dependence chain (the longest 'branch' in the parallel 'tree')

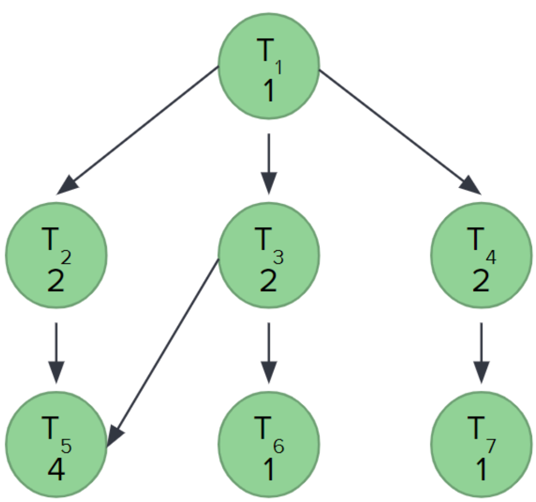
c) Does adding more processors affect the work or span?

Neither - both work and span are defined by a fixed number of processors (1 for work and infinity for
span) so adding more processors won't affect them

d) What is the total work and span of this task graph?

Total work = 1 + 2 + 2 + 2 + 4 + 1 + 1 = 13

Span = 1 + 2 + 4 = 7



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?

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