CSE 332 Spring 2026
Lecture 20: Parallel Prefix

Nathan Brunelle

http://www.cs.uw.edu/332

Which Data Structures are "Suitable" for Parallelism?

- For each data structure, can we write a parallel algorithm to sum all of its values that's *more efficient* than a sequential one?
  - Array
  - Linked List
  - Binary Tree

Reduction/Fold

- **Input**: array
- **Output:** single object (sum, max, min, parity, histogram, etc.)
- We "reduce" all elements in an array to a single item
- **Requirement:** operation done among elements is associative
  - $(x + y) + z = x + (y + z)$
  - min(min(x,y),z) = min(x, min(y,z))
  - parity(parity(x,y), z) = parity(x, parity(y,z))
- The "single item" can itself be complex
  - E.g. create a histogram of results from an array of trials

Find Max with ForkJoin

```
class MaxTask extends RecursiveTask<Integer> {
        int lo; int hi; int[] arr; // fields to know what to do
        SumTask(int[] a, int l, int h) { … }
        protected Integer compute(){// return answer
                if(hi – lo < SEQUENTIAL_CUTOFF) {  // base case
                        int ans = Integer.MIN_VALUE; // local var, not a field
                        for(int i=lo; i < hi; i++) {
                                ans = Math.max(ans, arr[i]);}
                        return ans;
                else {
                        MaxTask left = new MaxTask(arr,lo,(hi+lo)/2); // divide
                        MaxTask right= new MaxTask(arr,(hi+lo)/2,hi); // divide
                        left.fork(); // fork a thread and calls compute (conquer)
                        int rightAns = right.compute(); //call compute directly (conquer)
                        int leftAns = left.join(); // get result from left
                        return Math.max(rightAns, leftAns); // combine
                }
        }
}
```

Map

- **Input:** array(s)
- **Output:** array (of same size)
- **Requirement:** apply some function to individual array elements.
- Examples:
  - Vector addition:
    - sum[i] = arr1[i] + arr2[i]
  - Function application:
    - out[i] = f(arr[i]);

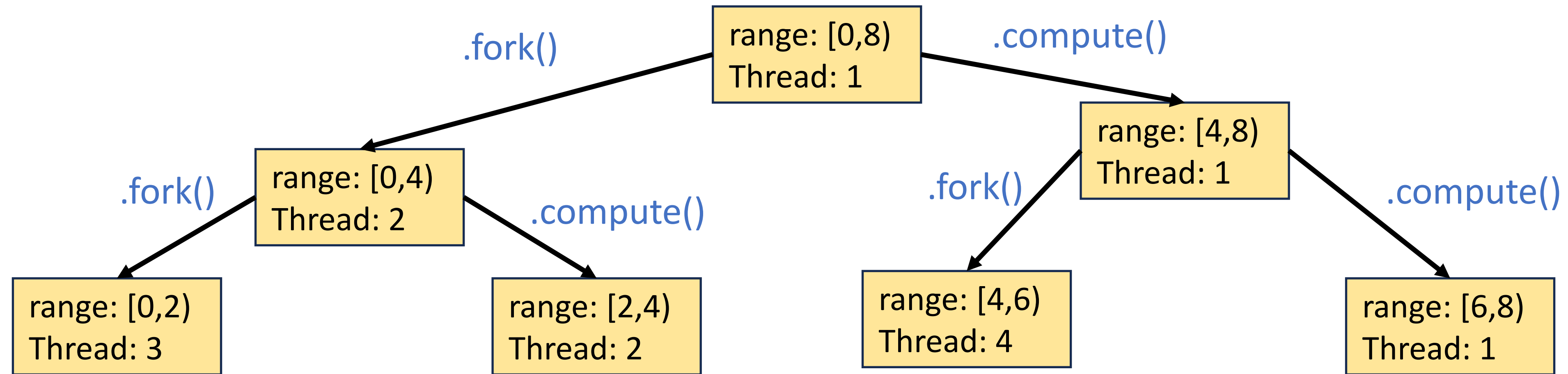
Vector Addition with ForkJoin

```
class AddTask extends RecursiveAction {
	int lo; int hi; int[] a; int[] b; int[] sum;
	AddTask(int[] a, int[] b, int[] sum, int l, int h) { … }
	protected void compute(){// return answer
		if(hi – lo < SEQUENTIAL_CUTOFF) {  // base case
			for(int i=lo; i < hi; i++) {
				sum[i] = a[i] + b[i];}
		else {
			AddTask left = new AddTask(a,b,sum,lo,(hi+lo)/2); // divide
			AddTask right= new AddTask(a,b,sum,(hi+lo)/2,hi); // divide
			left.fork(); // fork a thread and calls compute (conquer)
			right.compute(); //call compute directly (conquer)
			left.join(); // get result from left
			return; // combine
		}
	}
}
```

Function Application with ForkJoin

```
public String mapFunction(int x){…}
class MapTask extends RecursiveAction {
        int lo; int hi; int[] arr; String[] out
        MapTask(int[] arr, String[] out, int l, int h) { … }
        protected void compute(){// return answer
                if(hi – lo < SEQUENTIAL_CUTOFF) {  // base case
                        for(int i=lo; i < hi; i++) {
                                out[i] = mapFunction(arr[i]);}
                else {
                        MapTask left = new MapTask(arr,out,lo,(hi+lo)/2); // divide
                        MapTask right= new MapTask(arr,out,(hi+lo)/2,hi); // divide
                        left.fork(); // fork a thread and calls compute (conquer)
                        right.compute(); //call compute directly (conquer)
                        left.join(); // get result from left
                        return; // combine
                }
        }
}
```

ForkJoin Picture



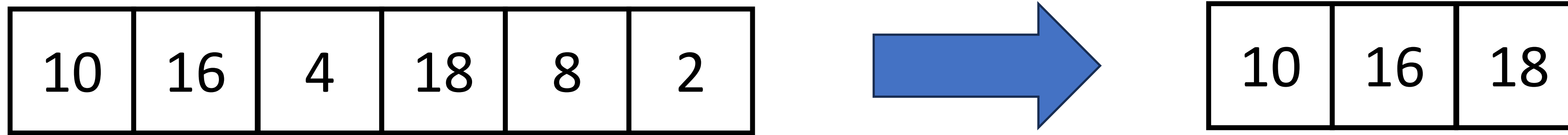
Map/Reduction Example

- **Task:** Multiply together the lengths of all the odd-length strings in array
  1. Apply map to convert the array of strings into an array of their lengths
  2. Then do a map on that array so each value maps to 1 if it's even and itself if it's odd
  3. Then do a reduction to multiply together that final result

- Note: You could do this in a single ForkJoin RecursiveTask
  - but "deconstructing" useful since some languages designed specifically for parallelism have Map/Reduce built in.
  - Map and Reduce are two from **a trio, with Pack/Filter** being the third

Pack/Filter

- Given an array of values and a Boolean function, return a new array which contains only elements that were “true”



$f(x) = x > 9$

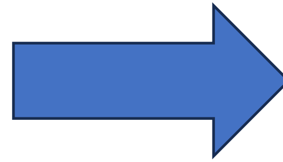
Question: Can we Pack/Filter in parallel?

Answer: yes, with help of parallel **Prefix Sum**

Prefix Sum

- Given an array, compute a new array where each index i is the sum of all values up to i

10	16	4	18	8	2



10	26	30	48	56	58

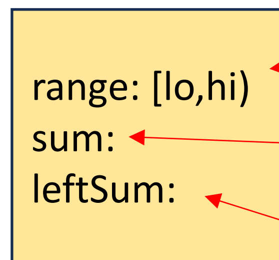
Later: will be useful for parallel packing/filtering!

Parallel Prefix Sum

- Algorithm will have two major parallel steps
  - Called a “two pass” parallel algorithm
- First step:
  - Create a tree data structure
- Second Step:
  - Use the tree to fill in the output array

Richard Ladner
Allen School Faculty

Tree Node:



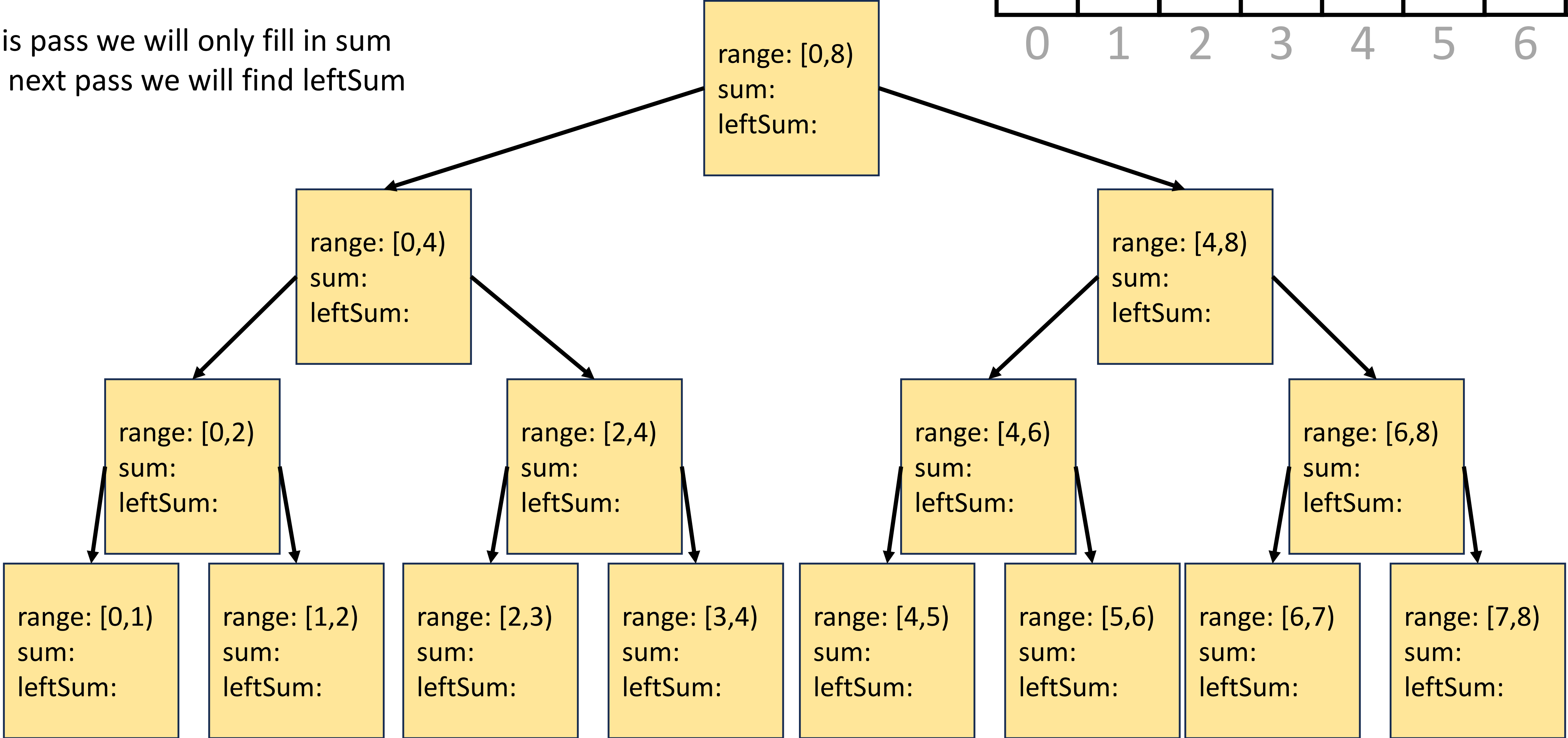
The “subproblem” this node represents
lower bound is inclusive, upper is exclusive

The sum of all values in the range

The sum of all values to the left of the range
i.e. in the range [0, lo)

Step 1: Using D&C
Create a Tree, Fill in sum

For this pass we will only fill in sum
In the next pass we will find leftSum



Step 1: Create a Tree, Fill in sum

10	16	4	18	8	2	14	9

4

range: [2,3)
sum: 4
leftSum:

8	2	14	9

10	16	4	18

range: [0,4)
sum: 48
leftSum:

range: [4,8)
sum: 33
leftSum:

range: [0,8)
sum: 81
leftSum:

range: [0,4)
sum: 48
leftSum:

range: [4,8)
sum: 33
leftSum:

- **Base Case**:
  - If the rand is smaller than the Sequential Cutoff, create a node for that range and find the sum sequentially
- **Divide**:
  - Split the list into two "sublists" of (roughly) equal length, create a thread for each sublist.
- **Conquer**:
  - Call **start()** for each thread to compute the left and right subtrees
- **Combine**:
  - Create parent node, connect to children, fill in sum

Step 1 pseudocode (create tree, compute sum)

BuildTree(arr)

1. **If** len(arr) < ℓ:
   1. Create new TreeNode Leaf
   2. Set Leaf.sum = sum of values in arr and return Leaf
2. **Else:**
   1. Divide arr in half into arr1 and arr2
   2. Conquer: TreeNode leftChild = **BuildTree**(arr1) in new thread, TreeNode rightChild = **BuildTree**(arr2) in this thread
   3. Wait for parallel computations to finish
   4. Combine: Create TreeNode parent, set parent.sum = leftChild.sum + rightChild.sum, parent.left = leftChild, parent.right = rightChild
   5. Return parent

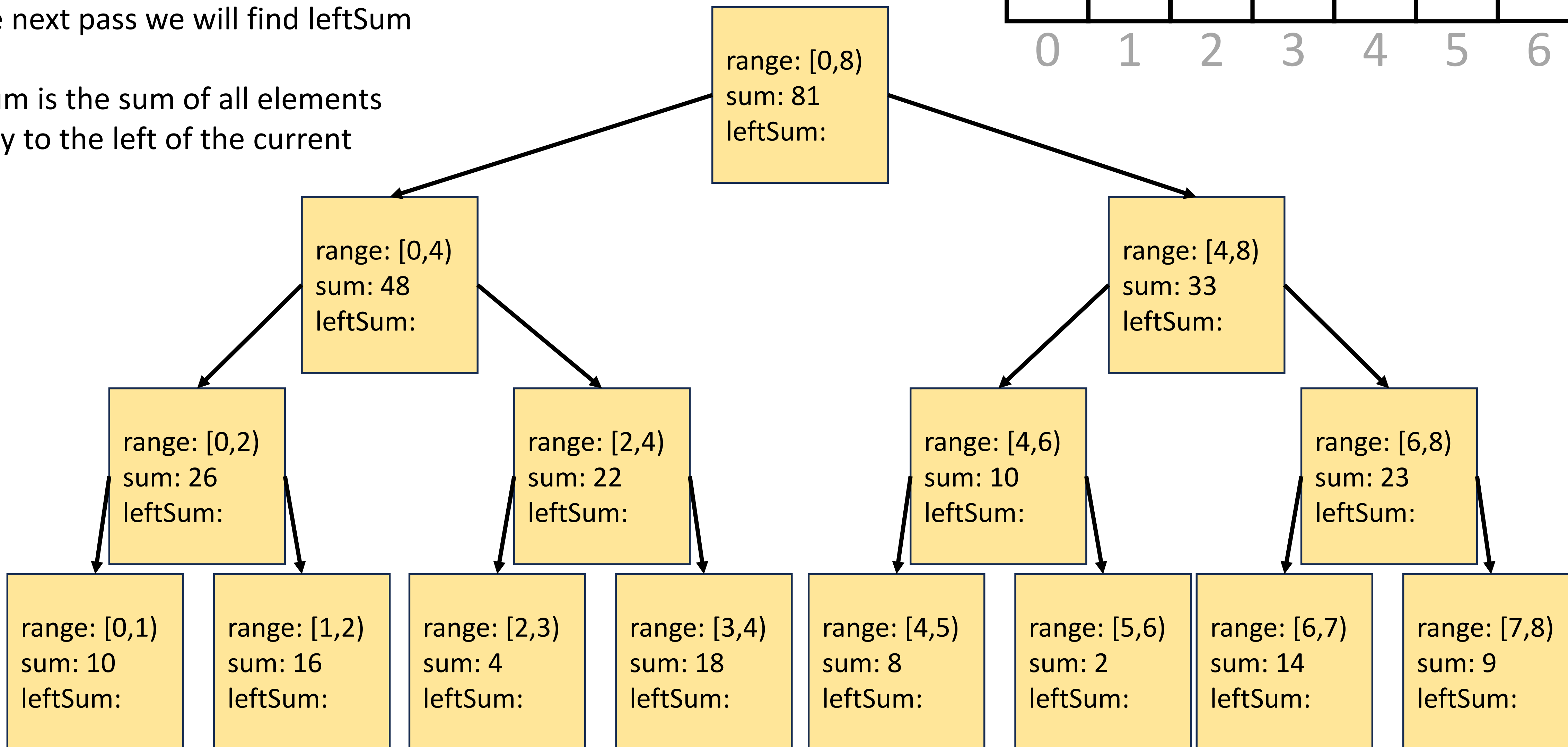
Step 1 Java Code

```
class BuildTreeTask extends RecursiveTask<PrefixSumNode> {
        int lo; int hi; int[] arr;
        public BuildTreeTask(int lo, int hi, int[] arr) {…}
        protected PrefixSumNode compute(){
                if(hi – lo < SEQUENTIAL_CUTOFF) {  // base case
                        int ans = 0; // local var, not a field
                        for(int i=lo; i < hi; i++)
                                ans += arr[i];
                        return new PrefixSumNode(lo, hi, ans); }
                else {
                        BuildTreeTask left = new BuildTreeTask(arr,lo,(hi+lo)/2);
                        BuildTreeTask right= new BuildTreeTask(arr,(hi+lo)/2,hi);
                        left.fork();
                        PrefixSumNode rightChild = right.compute();
                        PrefixSumNode leftChild = left.join();
                        int ans = rightChild.sum + leftChild.sum;
                        parent = new PrefixSumNode(lo, hi, ans);
                        parent.left = leftChild;
                        parent.right = rightChild;
                        return parent; }
        }
}
```

After Step 1

All sums filled in per node
In the next pass we will find leftSum

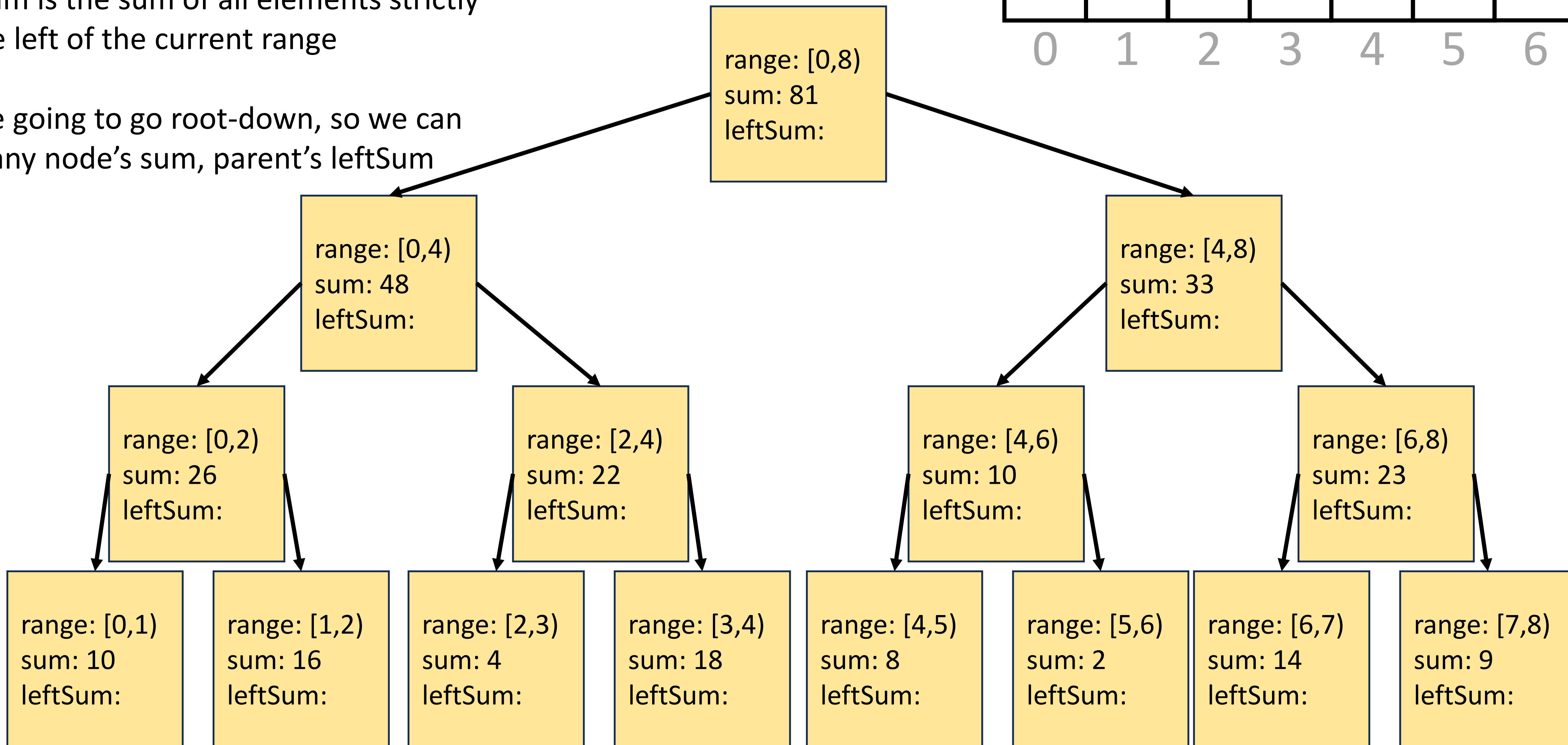
leftSum is the sum of all elements strictly to the left of the current range



Step 2: fill in leftSum and Output (1/4)

leftSum is the sum of all elements strictly to the left of the current range

We're going to go root-down, so we can use: any node's sum, parent's leftSum



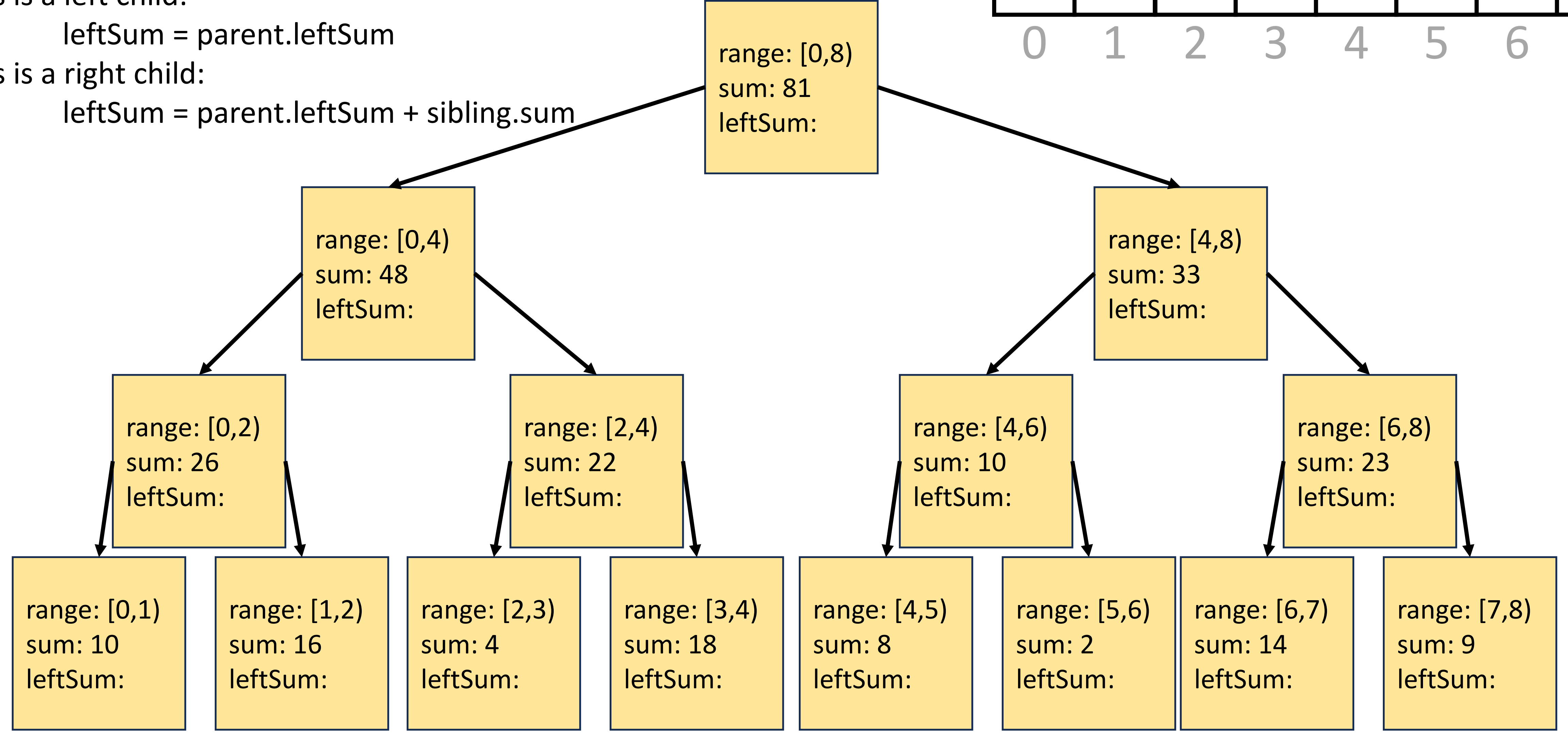
Step 2: fill in leftSum and Output (2/4)

If this is a left child:

leftSum = parent.leftSum

If this is a right child:

leftSum = parent.leftSum + sibling.sum



Step 2: fill in leftSum and Output (3/4)

If this is a left child:

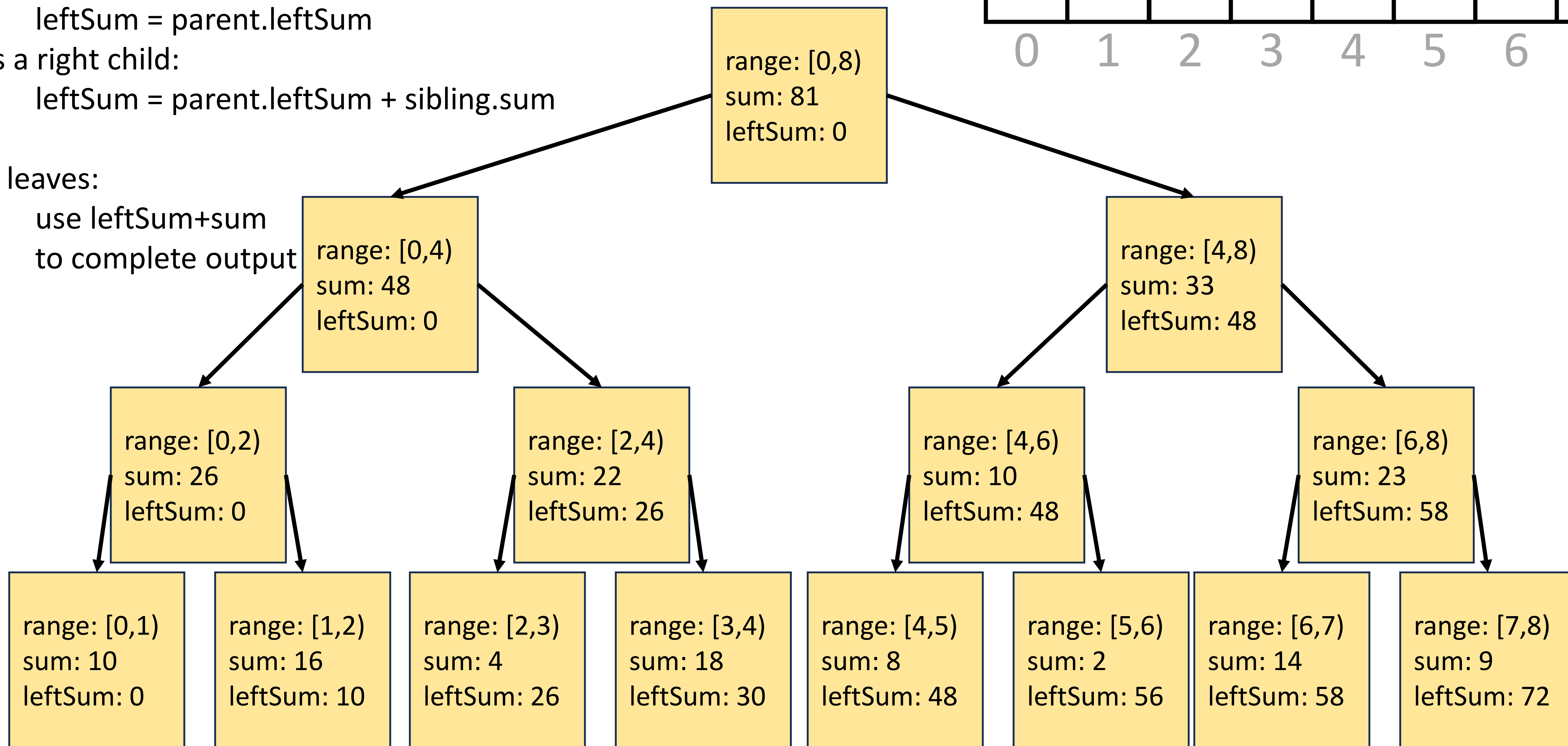
leftSum = parent.leftSum

If this is a right child:

leftSum = parent.leftSum + sibling.sum

For the leaves:

use leftSum+sum to complete output



Step 2: fill in leftSum and Output (4/4)

Input:	10	16	4	18	8	2	14	9
Output:	10	26	30	48	56	58	72	81
	0	1	2	3	4	5	6	7

If this is a left child:

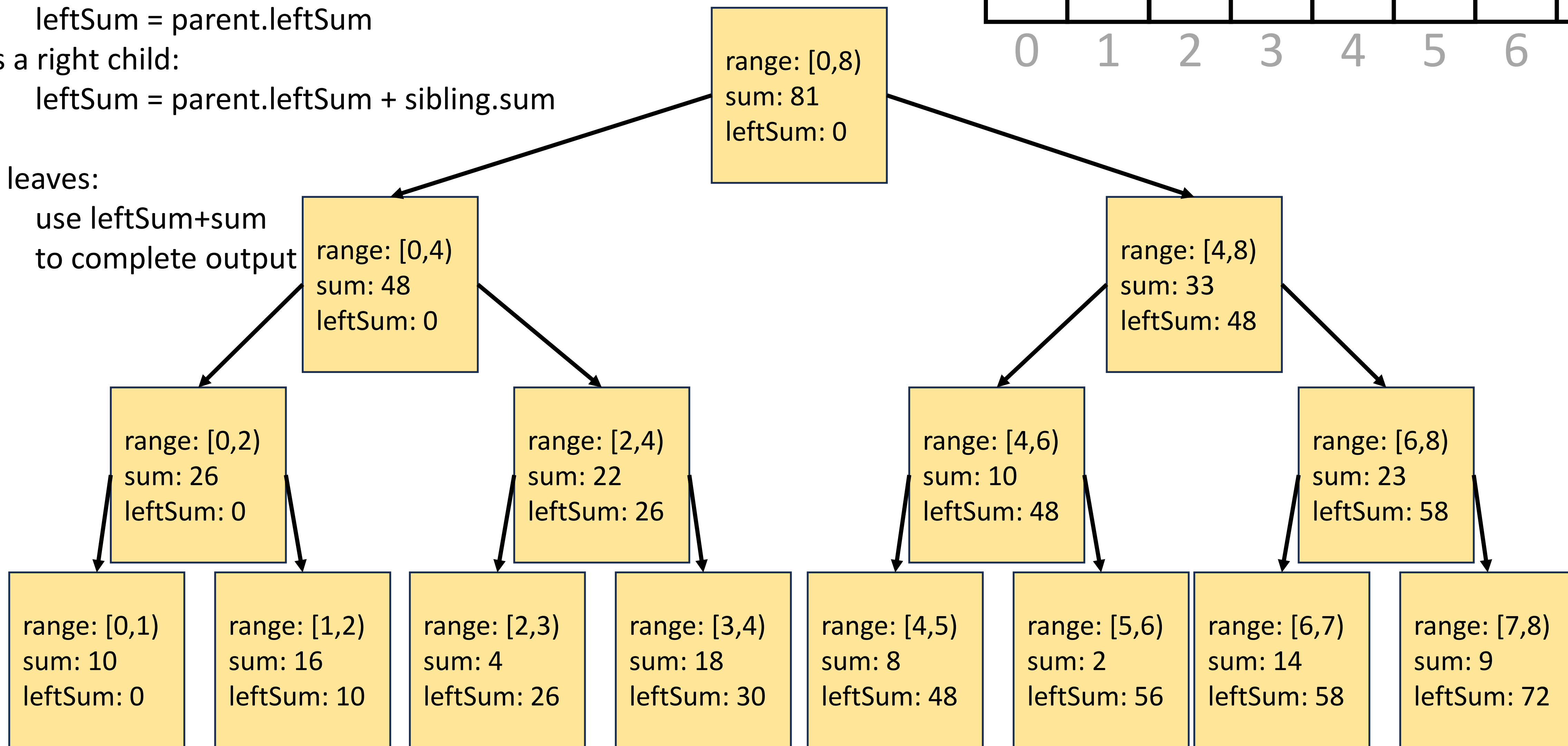
leftSum = parent.leftSum

If this is a right child:

leftSum = parent.leftSum + sibling.sum

For the leaves:

use leftSum+sum to complete output



Step 2 pseudocode (Compute Leftsum)

CompleteTree(Node curr)

1. **If** curr is a left child of curr.parent:
    1. Set curr.LeftSum = curr.parent.LeftSum
2. **Else:**
    1. Set curr.LeftSum = curr.parent.LeftSum + curr.sibling.Sum
3. **If** curr is not a leaf:
    1. <u>Divide and conquer:</u> call **CompleteTree**(curr.left) and **CompleteTree**(curr.right) in parallel threads
    2. <u>Wait</u> for parallel computations to finish
4. **Else (curr is a leaf)**:
    1. Set output[curr.lo] = curr.LeftSum + input[curr.lo]
    2. for i = curr.lo + 1 to curr.hi:
        1. Set output[i] = output[i-1] + input[i]

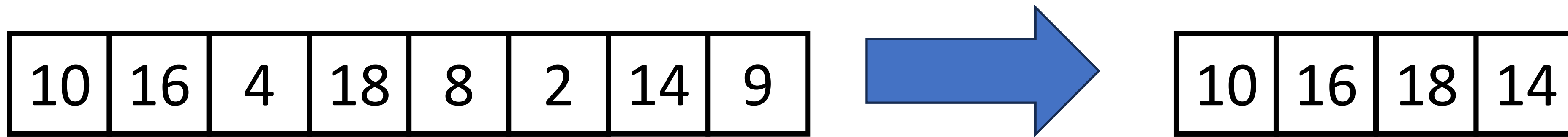
Base Case

Step 2 Java Code

```
class CompleteTree extends RecursiveAction {
        public CompleteTree(PrefixSumNode curr, PrefixSumNode parent, PrefixSumNode sibling, boolean isLeftChild, int[] output, int[] input){…}
        protected void compute(){
                if(parent == null) {curr.sumLeft = 0;} // if curr is the root
                else if(isLeftChild) {curr.sumLeft = parent.sumLeft;} // if curr is a left child
                else {curr.sumLeft = parent.sumLeft + sibling.sum;} // if curr is a right child
                if (curr.leftChild != null && curr.rightChild != null){  // if this isn’t a leaf
                        CompleteTree left = new CompleteTree(curr.leftChild, curr, curr.rightChild, true, output, input);
                        left.fork();
                        CompleteTree right = new CompleteTree(curr.rightChild, curr, curr.leftChild, false, output, input);
                        right.compute();
                        left.join();
                }
                else{ // if it is a leaf
                        output[curr.lo] = curr.sumLeft + input[curr.lo];
                        for(int i = curr.lo+1; i < curr.hi; i++){
                                output[i] = output[i-1] + input[i]
                        }
                }}}
```

Whew! Back to Pack/Filter

- Given an array of values and a Boolean function, return a new array which contains only elements that were “true”



$f(x) = x > 9$

Parallel Pack

Input:

10	16	4	18	8	2	14	9

, $f(x) = x > 9$

Output:

10	16	18	14

0 1 2 3 4 5 6 7

1. Do a **map** to identify the true elements

1	1	0	1	0	0	1	0

2. Do **prefix sum** on the result → count of true elements seen to the left of each position

1	2	2	3	3	3	4	4

3. Populate in the output in parallel:

10	16	18	14

3. Populate in the output in parallel:

Input:	10	16	4	18	8	2	14	9
Map Result:	1	1	0	1	0	0	1	0
Prefix Result:	1	2	2	3	3	3	4	4
Output:								

- Because the last value in the prefix result is 4, the length of the output is 4
- Each time there is a 1 in the map result, we want to include that element in the output
- If element i should be included, its position matches prefixResult[i]-1

In parallel: if mapResult[i] == 1:
output[prefixResult[i]-1] = input[i]

Step 3 Java Code

```
class PopulateTask extends RecursiveAction {
        int lo; int hi; int[] arr; int[] map; int[] prefix; int[] out;
        PopulateTask(int[] arr, int[] map, int[] prefix, int[] out, int l, int h) { … }
        protected void compute(){// return answer
                if(hi – lo < SEQUENTIAL_CUTOFF) {  // base case
                        for(int i=lo; i < hi; i++) {
                                if(map[i] == 1){
                                        out[prefix[i]-1]=arr[i];
                                }
                        }
                else {
                        PopulateTask left = new PopulateTask(arr, map, prefix, out, lo,(hi+lo)/2); // divide
                         PopulateTask right= new PopulateTask(arr, map, prefix, out, (hi+lo)/2,hi); // divide
                        left.fork(); // fork a thread and calls compute (conquer)
                        right.compute(); //call compute directly (conquer)
                        left.join(); // get result from left
                        return; // combine
                }
        }
}
```

Parallel Pack Java Code

```
public Boolean mapFunction(int x){…}
static final ForkJoinPool POOL = new ForkJoinPool();
public int[] parallelPrefixSum(int[] arr){
        int[] out = new int[arr.length];
        PrefixSumNode root = POOL.invoke(new BuildTreeTask(0, arr.length, arr));
        POOL.invoke(new CompleteTree(root, null, null, false, out, arr));
        return out;
}
public int[] parallelPack(int[] arr){
        int[] map = new int[arr.length];
        POOL.invoke(new MapTask(arr, map, 0, arr.length));
        int[] prefix = parallelPrefixSum(map);
        int[] pack = new int[prefix[prefix.length-1]];
        POOL.invoke(new PopulateTask(arr, map, prefix, pack, 0, arr.length));
        return pack;
}
```

Map/Reduction/Pack Example

- Multiply together the lengths of all of the odd-length strings in a given array
  - First, do a map to covert the array of strings into an array of their lengths
  - Then do a map on that array so each value maps to 1 if it’s even and itself if it’s odd
    - Alternatively, do a pack on the array to remove all even values
  - Then do a reduction to multiply together that final result