Deadlock & Parallel Algos

Intermediate Computer Systems Programming, Fall 2025

Instructor: Travis McGaha

TAs: Theodor Bulacovschi Ash Fujiyama

Administrivia

- Midterm Grading
  - Grades posted sometime on Thursday
  - Just waiting on a makeup exam
  - No talking about it yet please!
- HW08: Posted Yesterday
  - Have everything you need after this lecture
- Mid Semester Survey
  - Due End-of-day Nov 1st (Saturday)
- Check-in posted tomorrow

Lecture Outline

- **Liveness & Deadlock**
- Parallel Algorithms

Liveness

- Liveness: A set of properties that ensure that threads execute in a timely manner, despite any contention on shared resources.

- When `std::mutex::lock();` is called, the calling thread blocks (stops executing) until it can acquire the lock.
  - What happens if the thread can never acquire the lock?

Liveness Failure: Releasing locks

- If locks are not released by a thread, then other threads cannot acquire that lock

- See `release_locks.cpp`
  - Example where locks are not released once critical section is completed.

Liveness Failure: Deadlocks

- Consider the case where there are two threads and two locks
  - Thread 1 acquires lock1
  - Thread 2 acquires lock2
  - Thread 1 attempts to acquire lock2 and blocks
  - Thread 2 attempts to acquire lock1 and blocks

Neither thread can make progress ☹

- See `milk_deadlock.cpp`

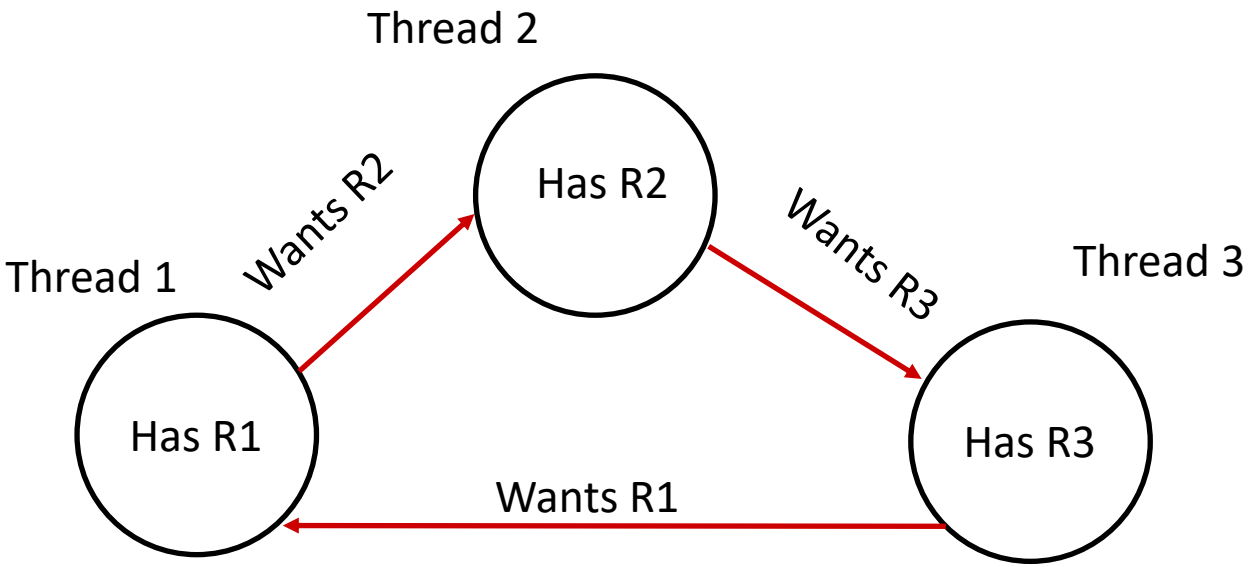
- Note: there are many algorithms for detecting/preventing deadlocks

Deadlock Definition

- A computer has multiple threads, finite resources, and the threads want to acquire those resources
  - Some of these resources require exclusive access
- A threads typically accumulate resources over time
  - If it fails to acquire a resource, it will (by default) wait until it is available before doing anything
- Deadlock: Cyclical dependency on resource acquisition so that none of them can proceed
  - Even if all unblocked threads release, deadlock will continue

Circular Wait Example

- A cycle can exist of more than just two threads:



How to avoid Deadlock

- Acquire all locks in the same order, all at once.
  - If you do this, then there is no way for a cycle to form

- std::scoped_lock can take in multiple locks and will acquire them in defined order to avoid deadlock. The order you pass the locks to scoped_lock doesn't matter.

- Most of the time you know which locks you need all at once.
  - If you don't, then you may not be able to acquire all locks in the same order.
  - Handling this gets complicated ☹ CIS 5480 dedicates 1.5-2 lectures on this
  - Put it simply: if you can't acquire them all at once, then when you acquire locks later you need to check to see if acquiring the lock would cause deadlock (by seeing if it would form a cycle in a graph like on the previous slide).

Liveness Failure: Mutex Recursion

- What happens if a thread tries to re-acquire a lock that it has already acquired?

- See `recursive_deadlock.cpp`

- By default, a mutex is not re-entrant.
  - The thread won't recognize it already has the lock, and block until the lock is released

Aside: Recursive Locks

- Mutex's can be configured so that you it can be re-locked if the thread already has locked it. These locks are called *recursive locks* (sometimes called *re-entrant locks*).

- Acquiring a lock that is already held will succeed
- To release a lock, it must be released the same number of times it was acquired

- Has its uses, but generally discouraged.

Lecture Outline

- Liveness & Deadlock
- **Parallel Algorithms**

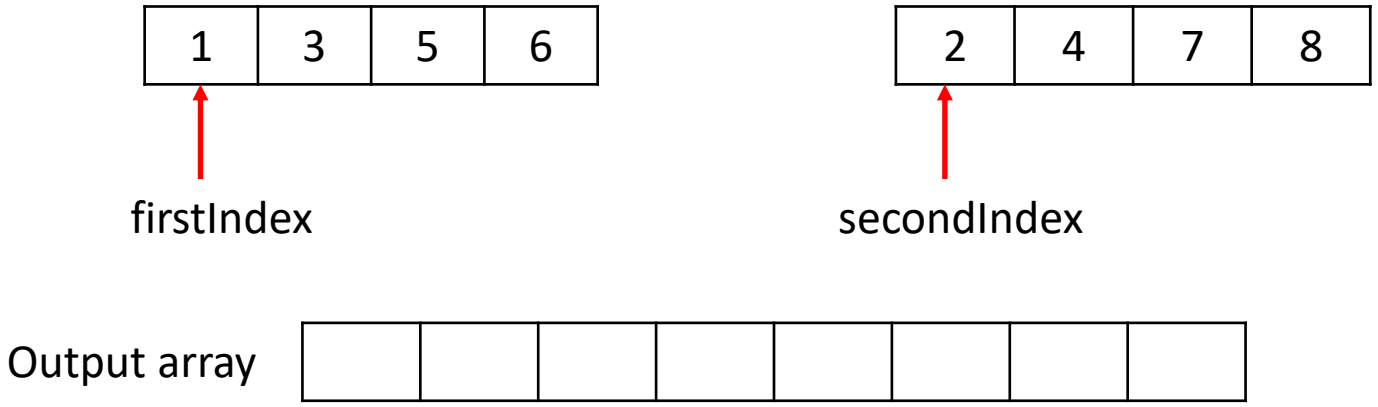
Parallel Algorithms

- One interesting applications of threads is for faster algorithms

- Common Example: Merge sort

Merge Sort: Core Ideas

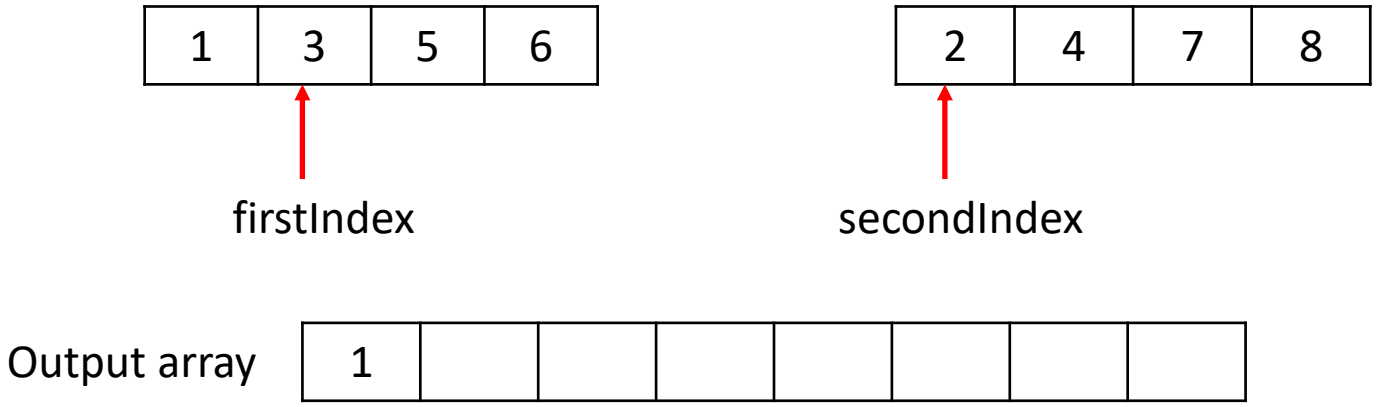
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



Merge Sort: Core Ideas

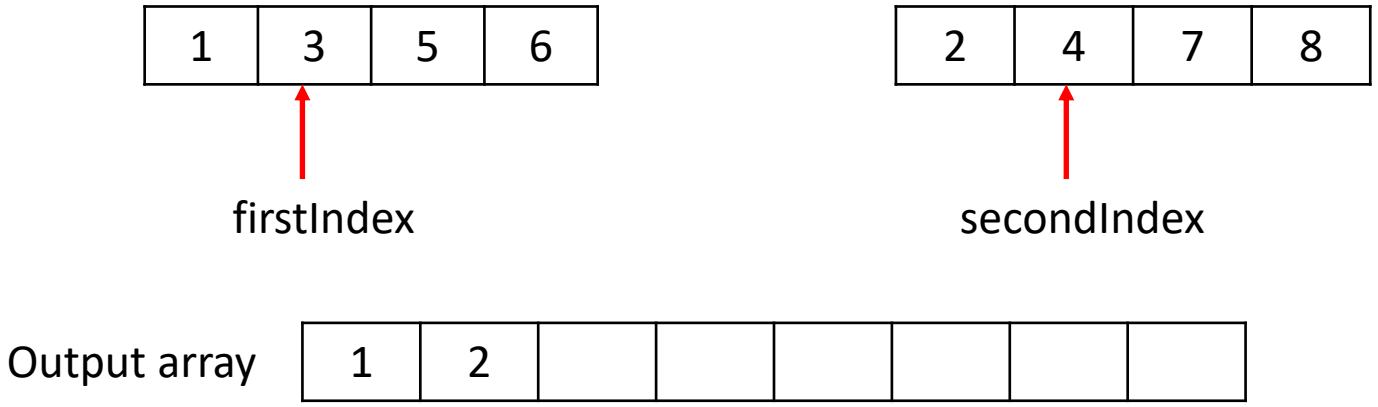
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

  - Consider the two sorted arrays:



Merge Sort: Core Ideas

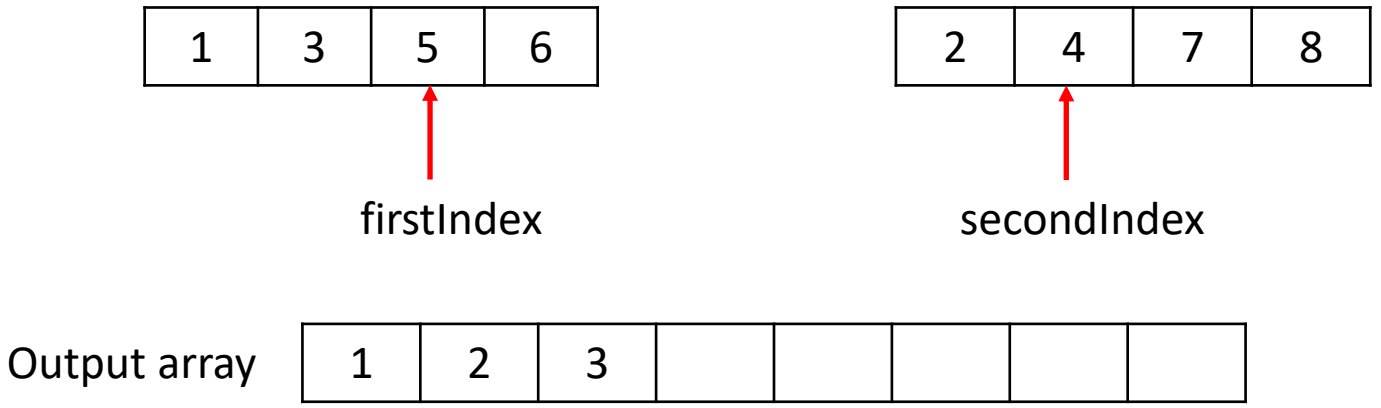
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



Merge Sort: Core Ideas

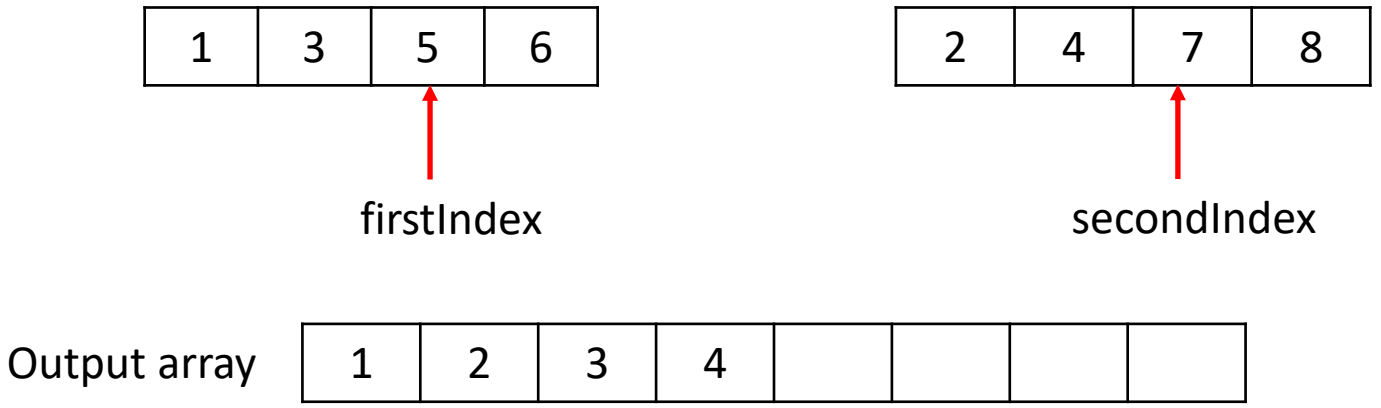
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

  - Consider the two sorted arrays:



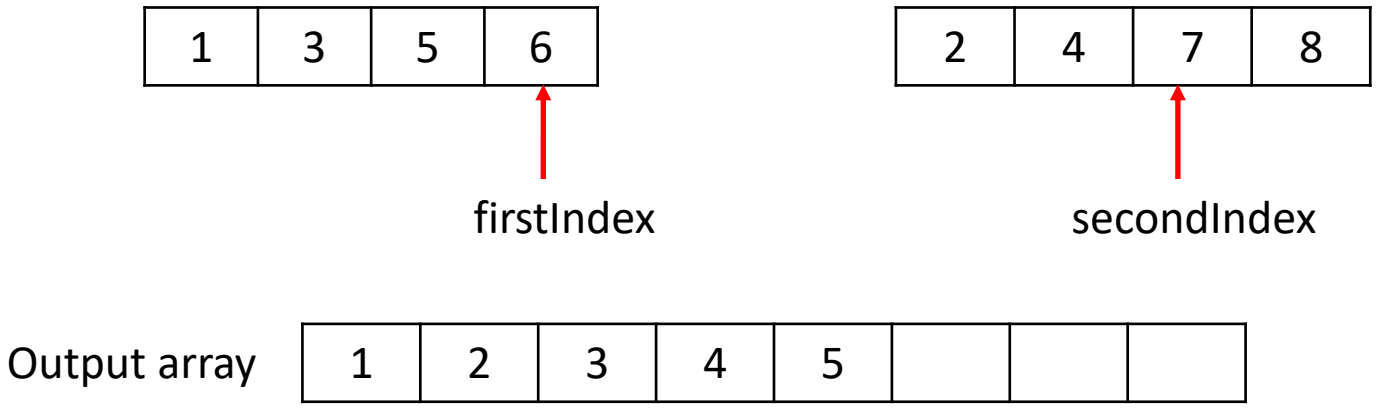
Merge Sort: Core Ideas

- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



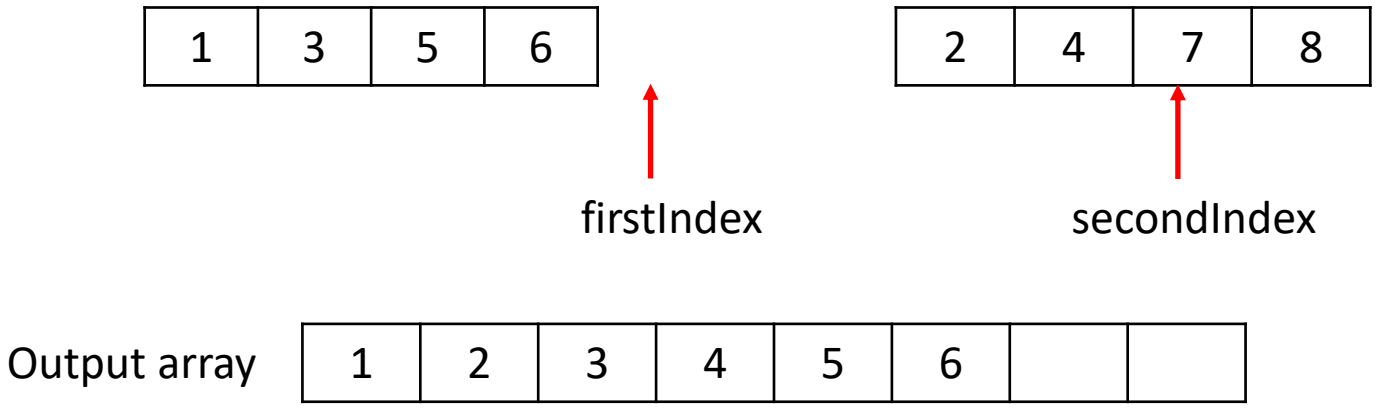
Merge Sort: Core Ideas

- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



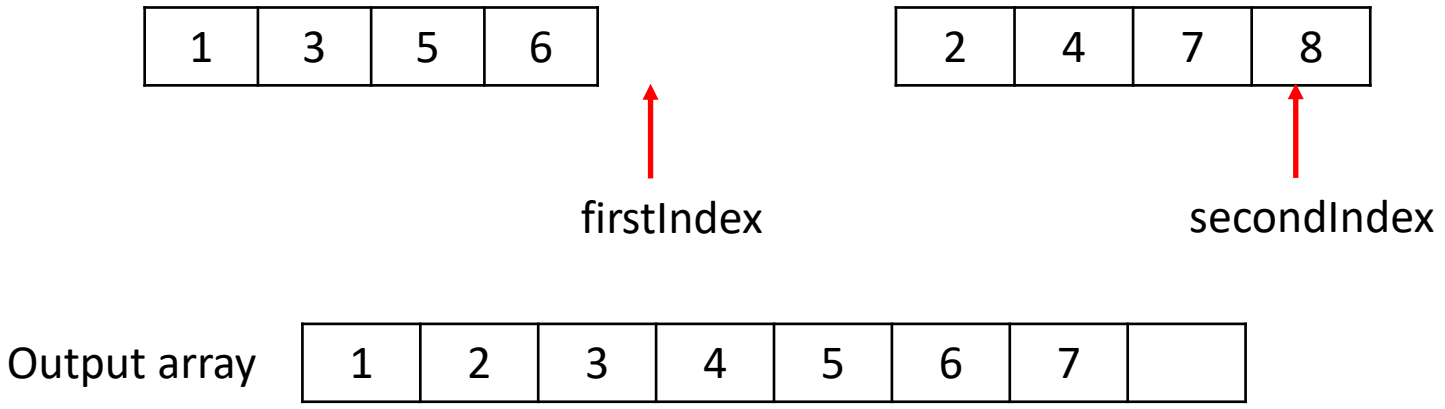
Merge Sort: Core Ideas

- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



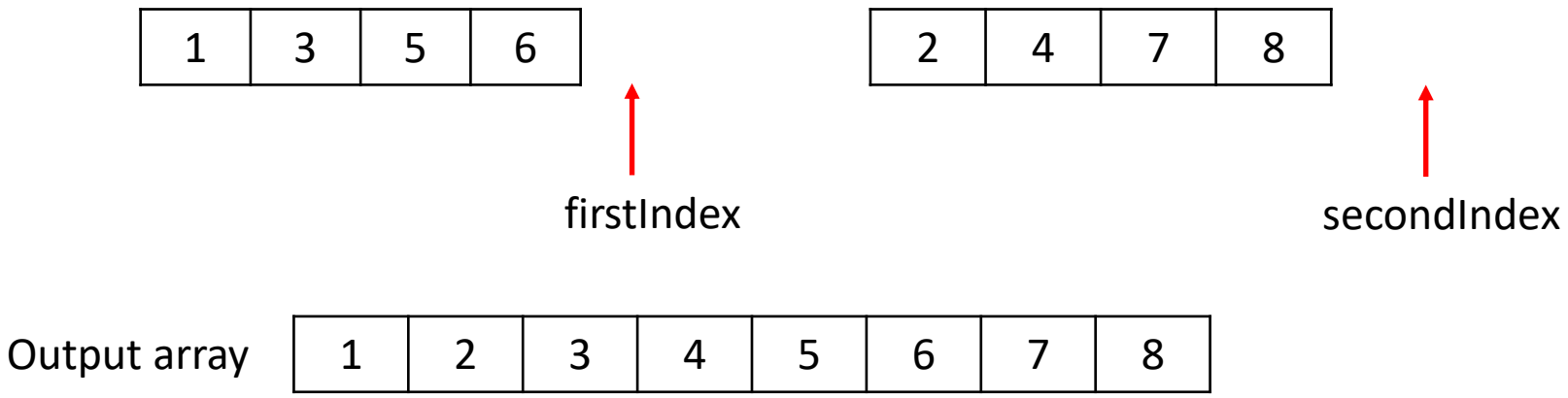
Merge Sort: Core Ideas

- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



Merge Sort: Core Ideas

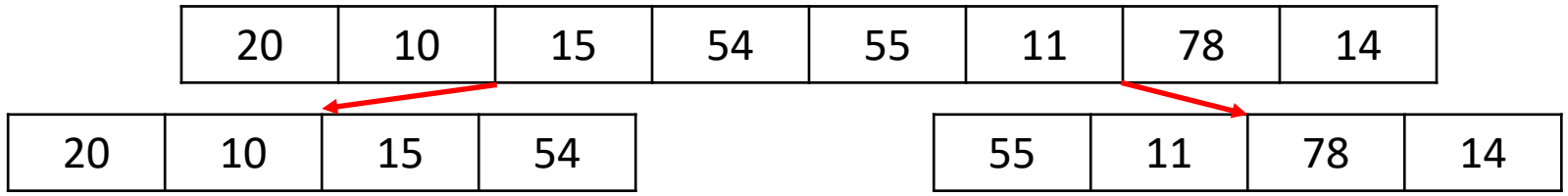
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



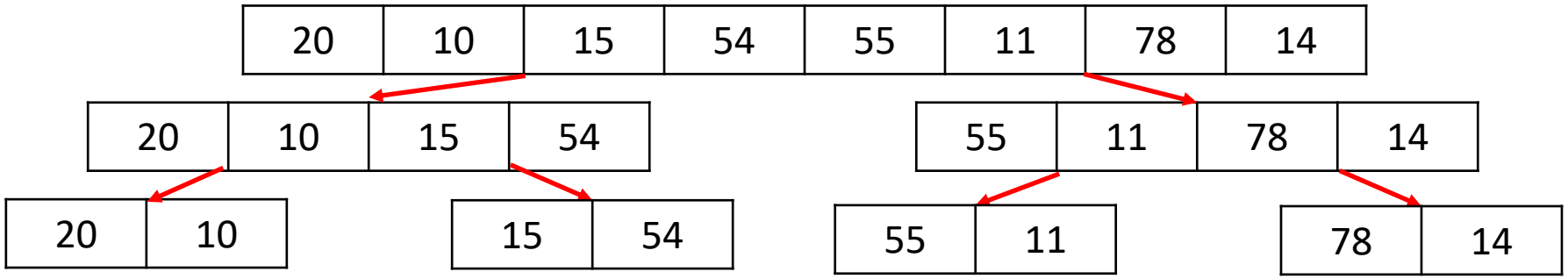
Merge Sort: High Level Example

20	10	15	54	55	11	78	14

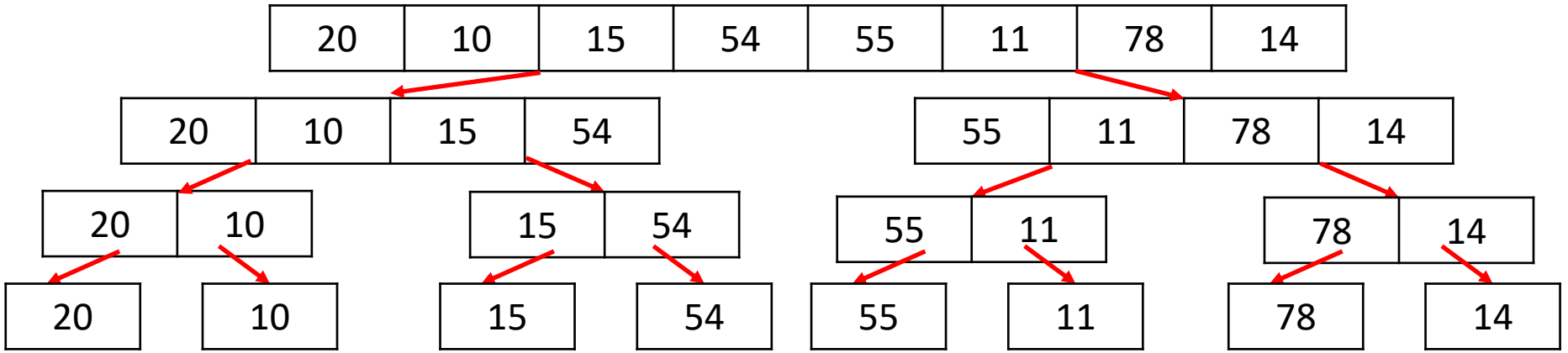
Merge Sort: High Level Example



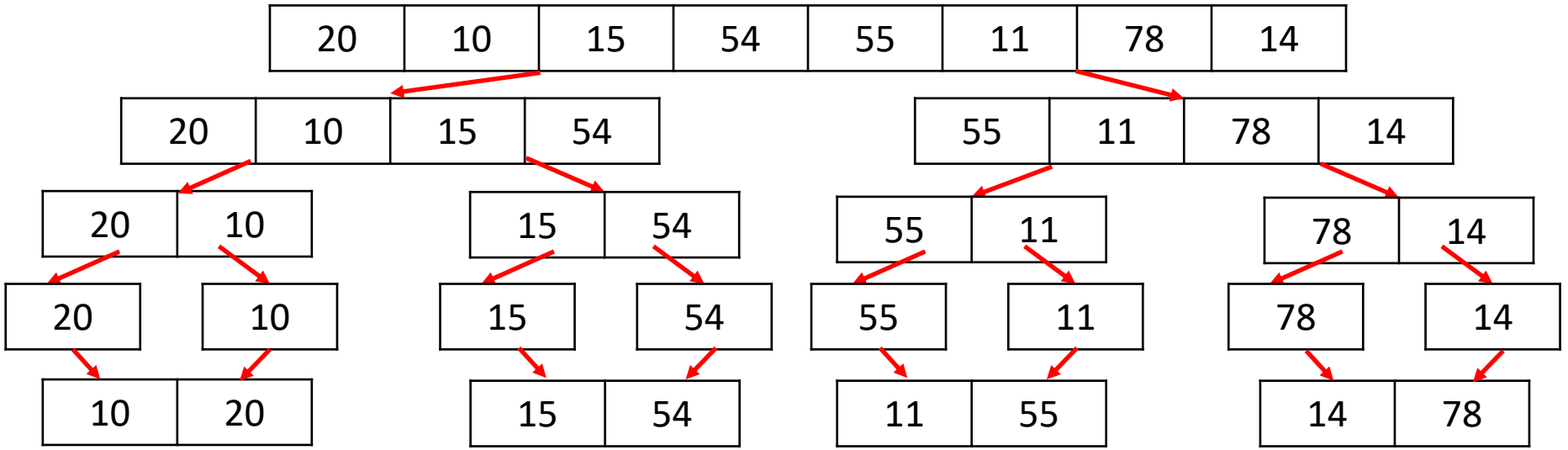
Merge Sort: High Level Example



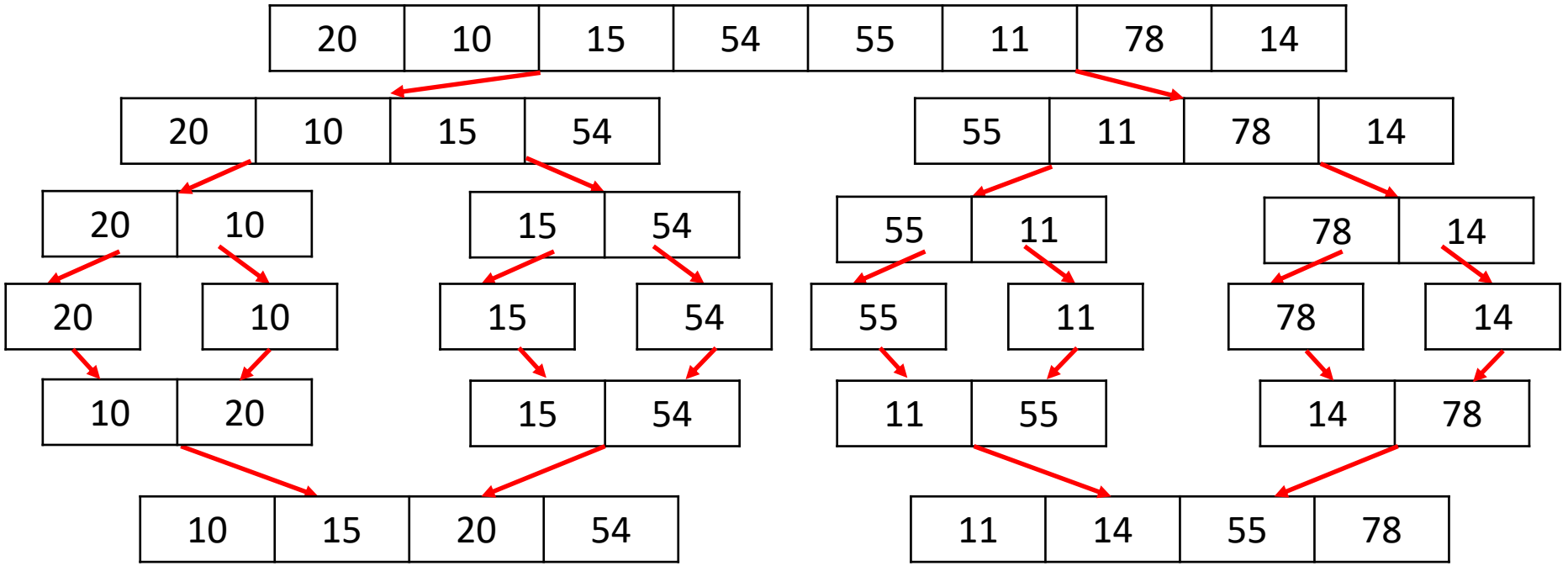
Merge Sort: High Level Example



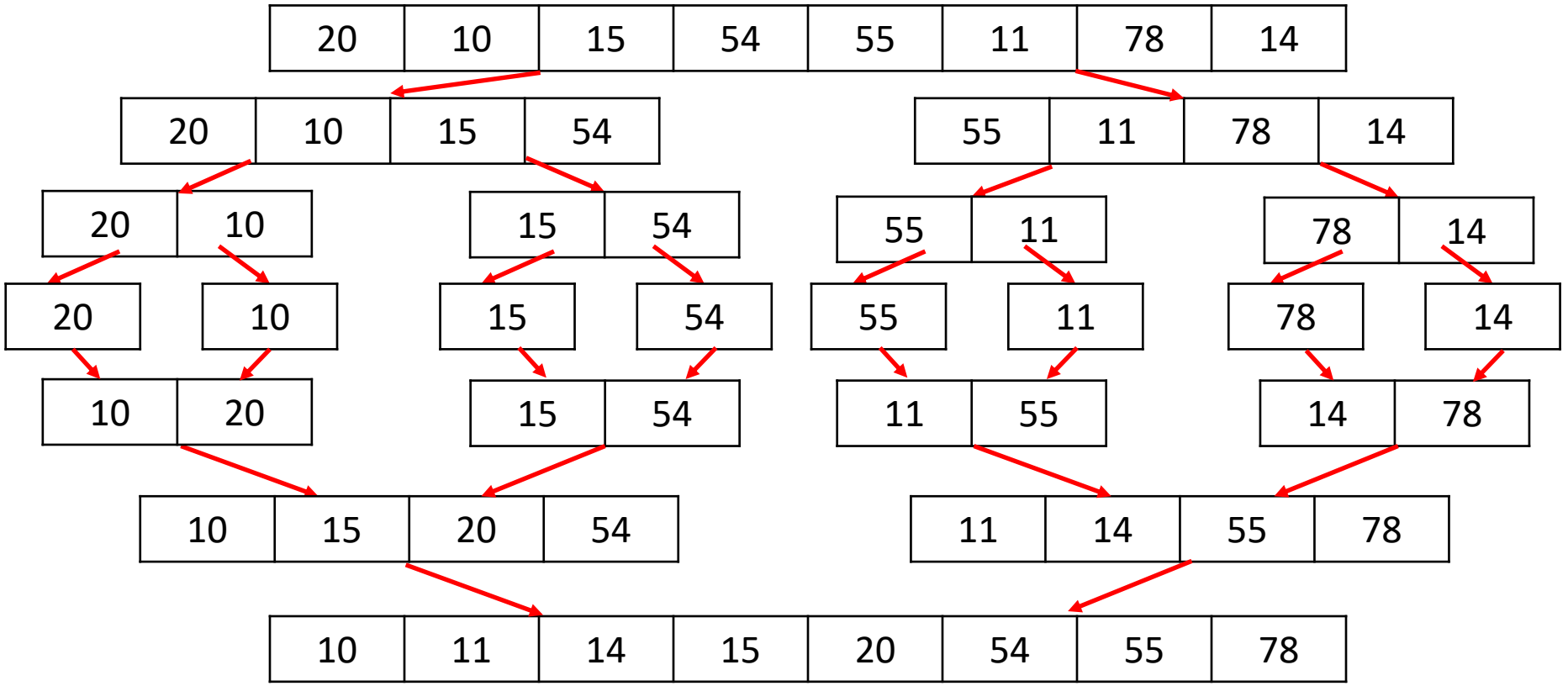
Merge Sort: High Level Example



Merge Sort: High Level Example



Merge Sort: High Level Example



Merge Sort Algorithmic Analysis

- Algorithmic analysis of merge sort gets us to O(n * log(n)) runtime.

```
void merge_sort(int[] arr, int lo, int hi) {
  // lo high start at 0 and arr.length respectively
  int mid = (lo + hi) / 2;
  merge_sort(arr, lo, mid);  // sort the bottom half
  merge_sort(arr, mid, hi);  // sort the upper half

  // combine the upper and lower half into one sorted
  // array containing all eles
  merge(arr[lo : mid], arr[mid : hi]);
}
```

- We recurse $\log_2(N)$ times, each recursive "layer" does O(N) work

Merge Sort Algorithmic Analysis

- We can use threads to speed this up:

```
void merge_sort(int[] arr, int lo, int hi) {
  // lo high start at 0 and arr.length respectively
  int mid = (lo + hi) / 2;

  // sort bottom half in parallel
  std::jthread thd(merge_sort,arr, lo, mid);
  merge_sort(arr, mid, hi);  // sort the upper half

  thd.join(); // join the thread that did bottom half

  // combine the upper and lower half into one sorted
  // array containing all eles
  merge(arr[lo : mid], arr[mid : hi]);
}
```

  - Now we are sorting both halves of the array in parallel!

Discuss

- We can use threads to speed this up:

```
void merge_sort(int[] arr, int lo, int hi) {
  // lo high start at 0 and arr.length respectively
  int mid = (lo + hi) / 2;

  // sort bottom half in parallel
  std::jthread thd(merge_sort,arr, lo, mid);
  merge_sort(arr, mid, hi);  // sort the upper half

  thd.join(); // join the thread that did bottom half

  // combine the upper and lower half into one sorted
  // array containing all eles
  merge(arr[lo : mid], arr[mid : hi]);
}
```

  - Now we are sorting both halves of the array in parallel!
  - **How long does this take to run?**
  - **How much work is being done?**

Parallel Algos:

Will not test you on this

- We can define **T(n)** to be the running time of our algorithm

- We can split up our work between two parts, the part done sequentially, and the part done in parallel
  - T(n) = sequential_part + parallel_part
  - T(n) = O(n) *merging* + T(n/2) *sort half the array*
    - This is a recursive definition

- If we start recurring…
  - T(n) = O(n) + O(n/2) + T(n/4)
  - T(n) = O(n) + O(n/2) + O(n/4) + T(n/8)

Parallel Algos:

Will not test you on this

- If we start recurring…
  - T(n) = O(n) + O(n/2) + T(n/4)
  - T(n) = O(n) + O(n/2) + O(n/4) + T(n/8)
  - …
  - Eventually we stop, there is a limit to the length of the array.
    And we can say an array of size 1 is already sorted, so T(1) = O(1)

- This approximates to T(n) = ~2 * O(n) = O(n)
  - This parallel merge sort is O(n), but there are further optimizations that can be done to reach ~O(log(n))

- There is a lot more to parallel algo analysis than just this, I am just giving you a sneak peek

Amdahl's Law

- For most algorithms, there are parts that parallelize well and parts that don't. This causes adding threads to have diminishing returns
  - (even ignoring the overhead costs of creating & scheduling threads)

- Consider we have some parallel algorithm $T_1 = 1$
  - The 1 subscript indicates this is run on 1 thread
  - we define the work for the entire algorithm as 1

- We define S as being the part that can be parallelized
  - $T_1 = S + (1 - S)$ *// (1-S) is the sequential part*

Amdahl's Law

- For running on one thread:
  - $T_1 = (1 - S) + S$

- If we have P threads and perfect linear speedup on the parallelizable part, we get
  - $T_P = (1\text{-}S) + \frac{S}{P}$

- Speed up multiplier for P threads from sequential is:
  - $\frac{T_1}{T_p} = \frac{1}{1-S+\frac{S}{P}}$

Amdahl's Law

- Let's say that we have 100000 threads (P = 100000) and our algorithm is only 2/3 parallel? (s = 0.6666..)
  - $\frac{T_1}{T_p} = \frac{1}{1-0.6666+\frac{0.6666}{100000}} = 2.9999\ times\ faster\ than\ sequential$

- What if it is 90% parallel? (S = 0.9):
  - $\frac{T_1}{T_p} = \frac{1}{1-0.9+\frac{0.9}{100000}} = 9.99\ times\ faster\ than\ sequential$

- What if it is 99% parallel? (S = 0.99):
  - $\frac{T_1}{T_p} = \frac{1}{1-0.99+\frac{0.99}{100000}} = 99.99\ times\ faster\ than\ sequential$

Limitation: Hardware Threads

- These algorithms are limited by hardware.
- Number of Hardware Threads: The number of threads can genuinely run in parallel on hardware

- We may be able to create a huge number of threads, but only run a few (e.g. 4) in parallel at a time.

- Can see this information in with **`lscpu`** in bash or **`std::thread::hardware_concurrency()`**
  - A computer can have some number of CPU sockets
  - Each CPU can have one or more cores
  - Each Core can run 1 or more threads

Limitations: Other Hardware

- This algorithm analysis assumes we are spending time purely in the CPU
- It doesn't account for threads blocking on I/O or other hardware.

Limitation: Threads, Locks & Synchronization Overhead

- It didn't apply in this example, but creating threads, and acquiring locks introduces overhead
- Can use a threadpool to minimize thread creation overhead
- Locks and synchronization is generally minimized as much as possible.
- Goal: Give each thread an independent piece of work to do and then look at the results after it is done.

Transform / Map

- Consider the map (called transform in C++) higher order function
  - Given a sequence, returns the result of calling a function on each element in the input sequence.

```
Given a function "func" and an input vector:
  {a, b, c, d, e, f, ..., z}

Then the returned value is:
  {func(a), func(b), func(c), func(d), func(e), func(f), ..., func(z)}
```

- How do we parallelize this?

Embarrassingly Parallel

- Problems that take little to no effort to distribute the work across threads and merge the results back in together.
- There is no dependency between threads (no communication or information needs to be shared), except for the final step of combining the final results together.
- Transform is embarrassingly parallel since we just split up the work across threads. Each thread's work is "pure", and doesn't depend on the state of other thread's work.

Reduce

Example of a reduce:
given a vector<int>
find the sum of all elements.

- Consider the reduce higher order function
  - Given a sequence, reduce the elements into a single result.
    Also given a function that takes two elements and combines them

```
Given a function "func" and an input vector:
  {a, b, c, z}

Then the returned value is equivalent to:
  func(func(func(a, b), c), z)
```

- You can assume func is associative:

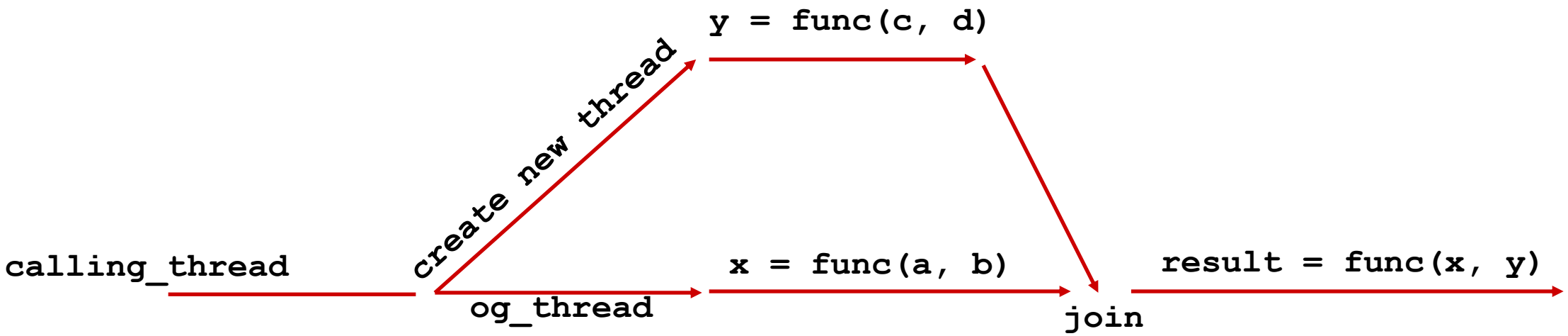
```
func(func(x, y), z) == func(x, func(y, z))
```

- How do we parallelize this?
  - How can we split up the work?

Reduce (Small Example)

- You can assume func is associative: `func(func(x, y), z) == func(x, func(y, z))`

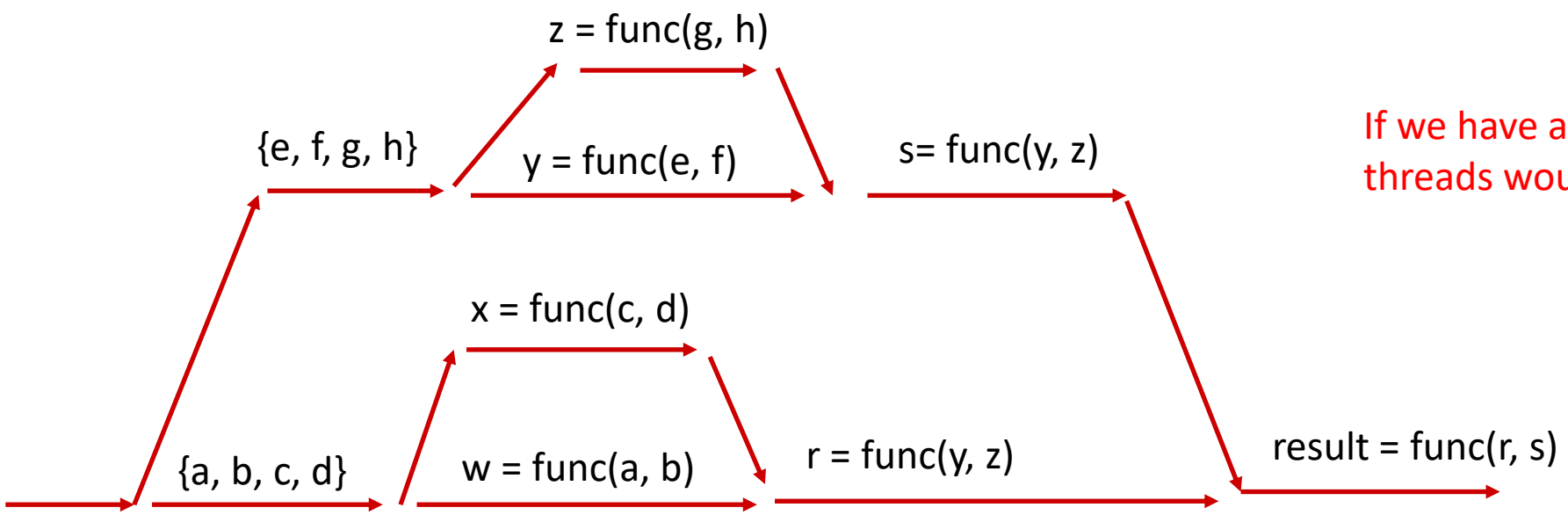
- Consider the input vector: **{a, b, c, d}**
- the result should be **{func(func(func(a, b), c), d)}**
- which is equal to **{func(func(a, b), func(c, d))}**
- We can divide up the work!



Reduce (Extrapolating to larger examples)

- You can assume func is associative: `func(func(x, y), z) == func(x, func(y, z))`

- Consider the input vector: `{a, b, c, d, e, f, g, h}`
- How do we split up work across more than 2 threads?



How many threads did this create?

3

If we have an input of size N, how many threads would we benefit from using?

~$\log_2(n)$

Parallel Prefix Sum

Credit to UW CSE 332 for teaching me this and how to teach this

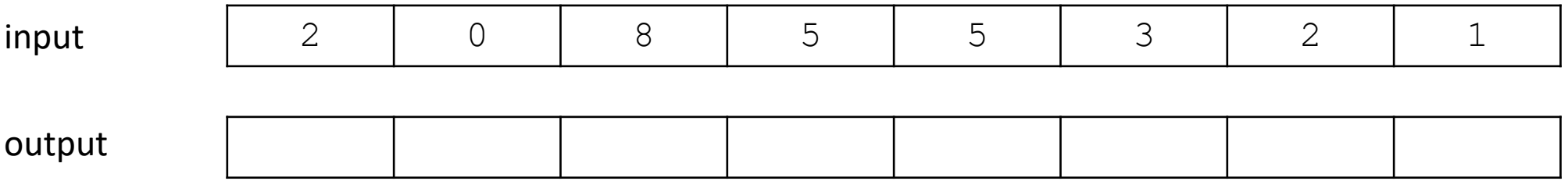
- Prefix sum, given an input array of integers, calculate a new vector which is the prefix sum. Each element at index `i` is the **`sum(input[0 ... i])`**
  - Given: `{2, 0, 8, 5, 5, 3, 2, 1}`
  - Result: `{2, 2, 10, 15, 20, 23, 25, 26}`

- How do we parallelize this? Values seem very dependent on each other…
  - Sort of similar to merge sort, we split it up across ~n threads
  - But we do it in two passes!

Parallel Prefix Sum

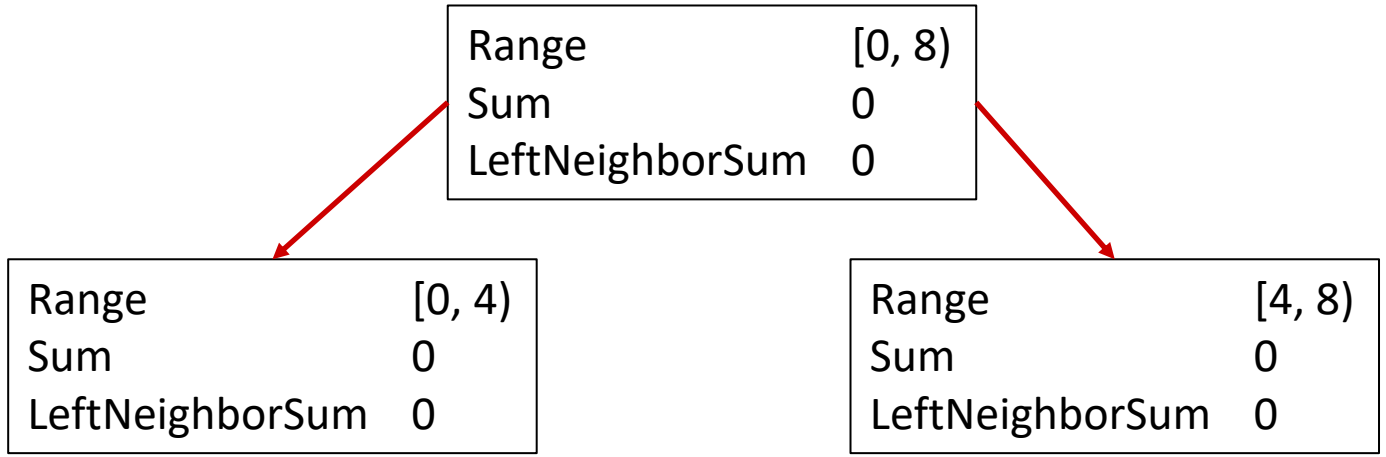
Credit to UW CSE 332 for teaching me this and how to teach this

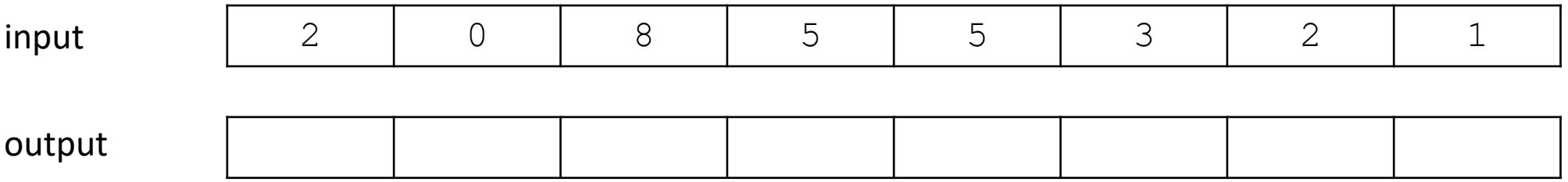
Range	[0, 8)
Sum	0
LeftNeighborSum	0



Parallel Prefix Sum

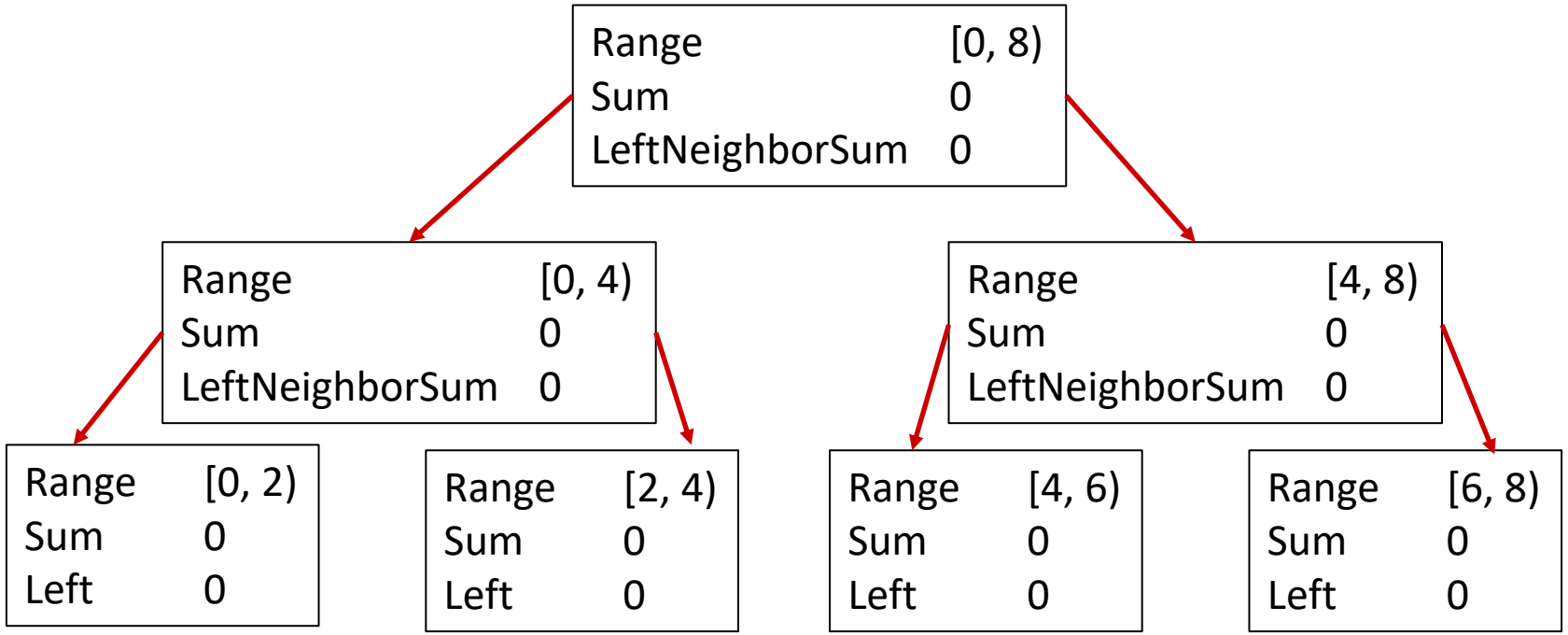
Credit to UW CSE 332 for teaching me this and how to teach this

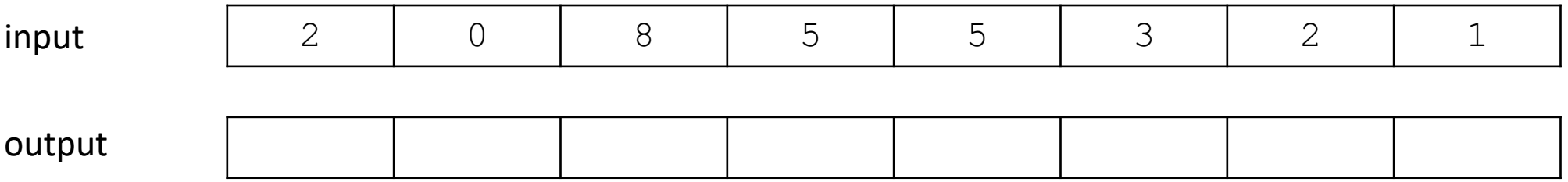




Parallel Prefix Sum

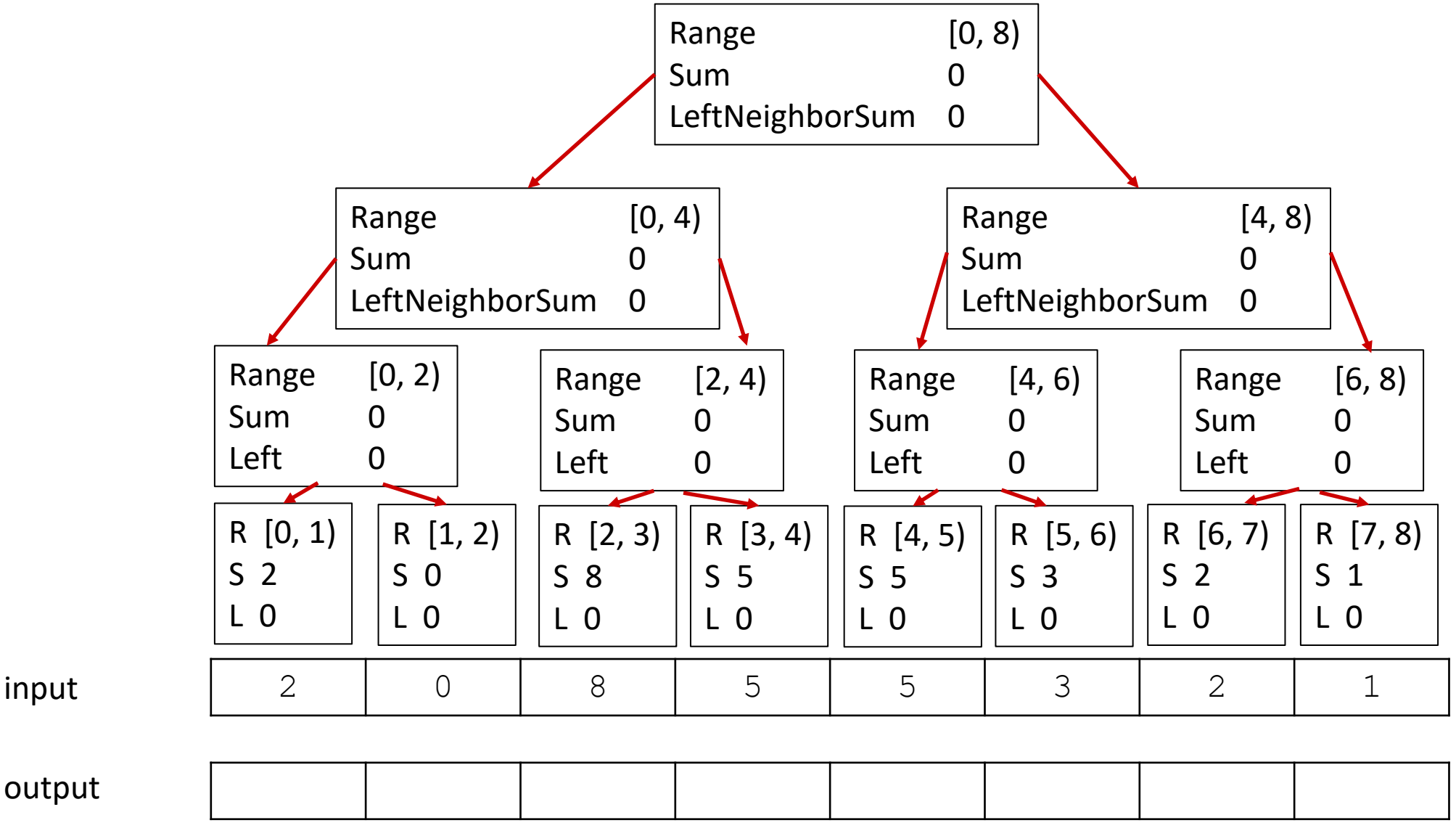
Credit to UW CSE 332 for teaching me this and how to teach this





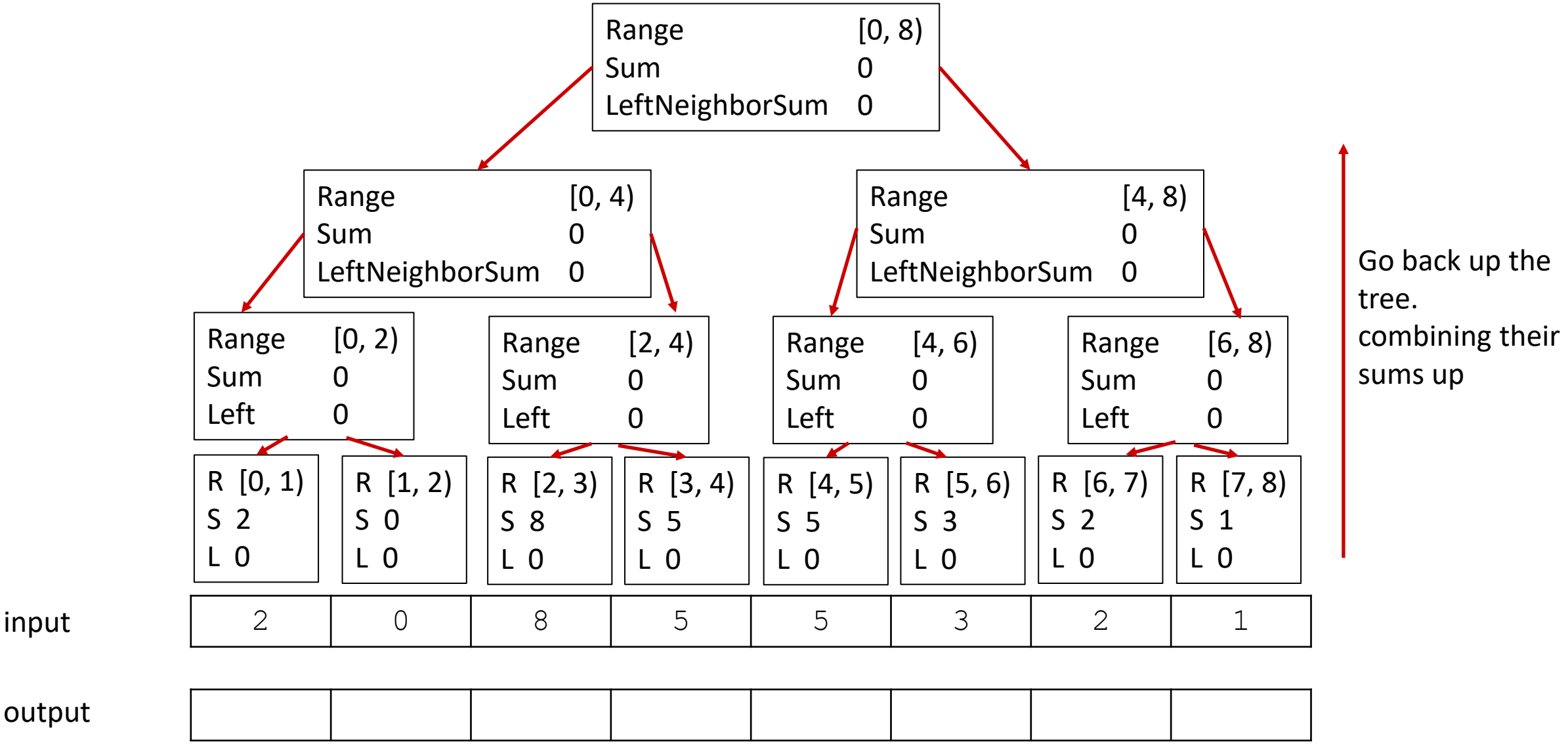
Parallel Prefix Sum

Credit to UW CSE 332 for teaching me this and how to teach this



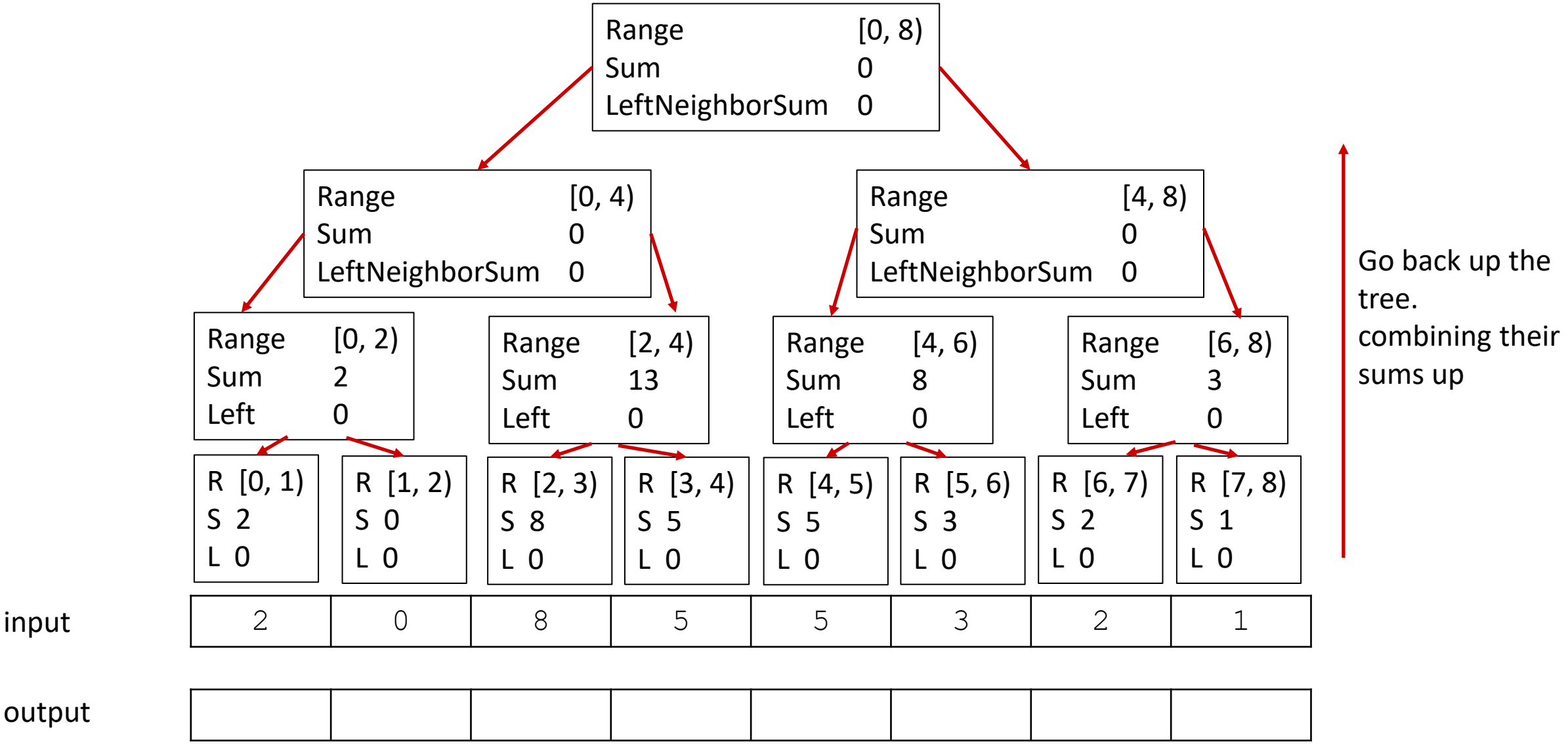
Parallel Prefix Sum

Credit to UW CSE 332 for teaching me this and how to teach this



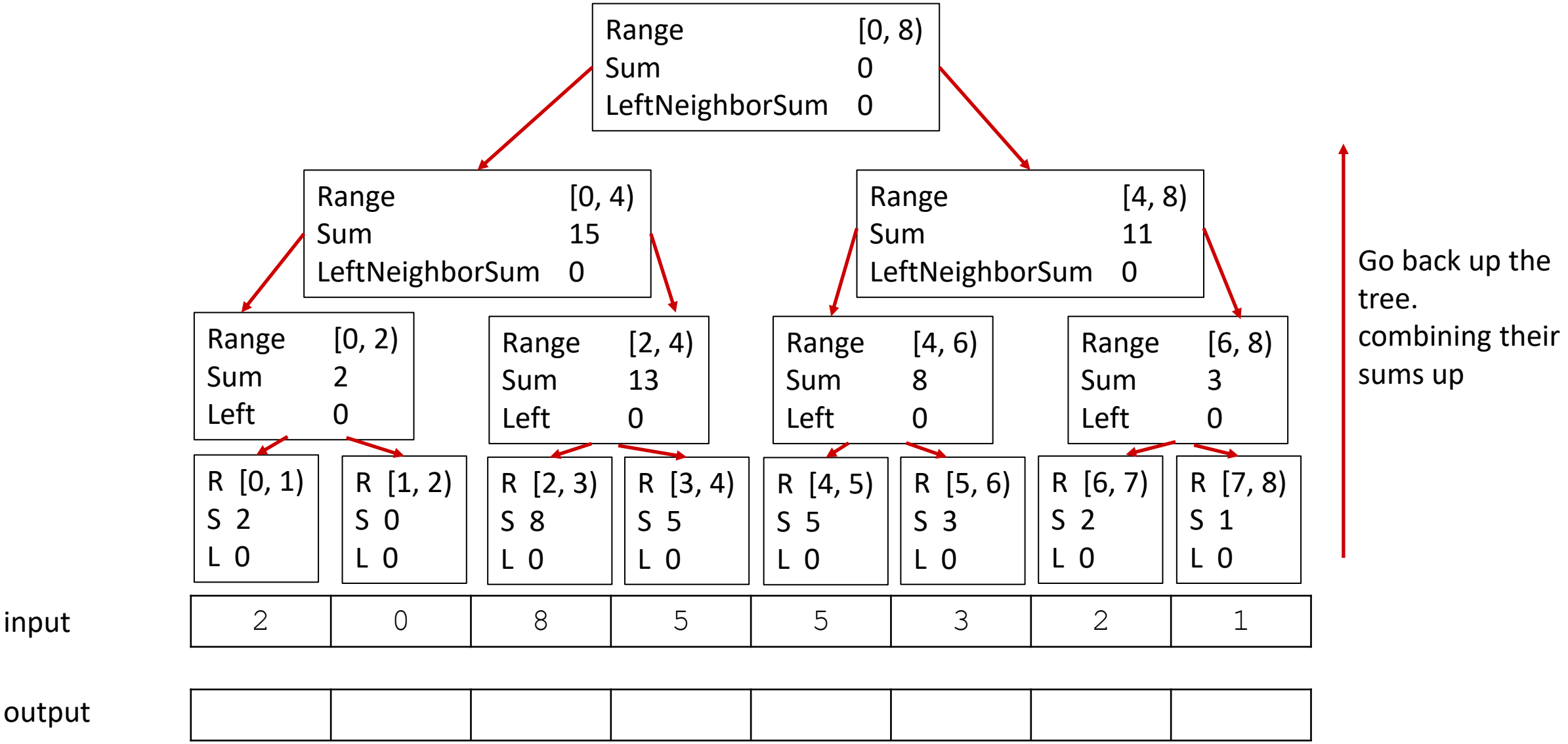
Parallel Prefix Sum

Credit to UW CSE 332 for teaching me this and how to teach this



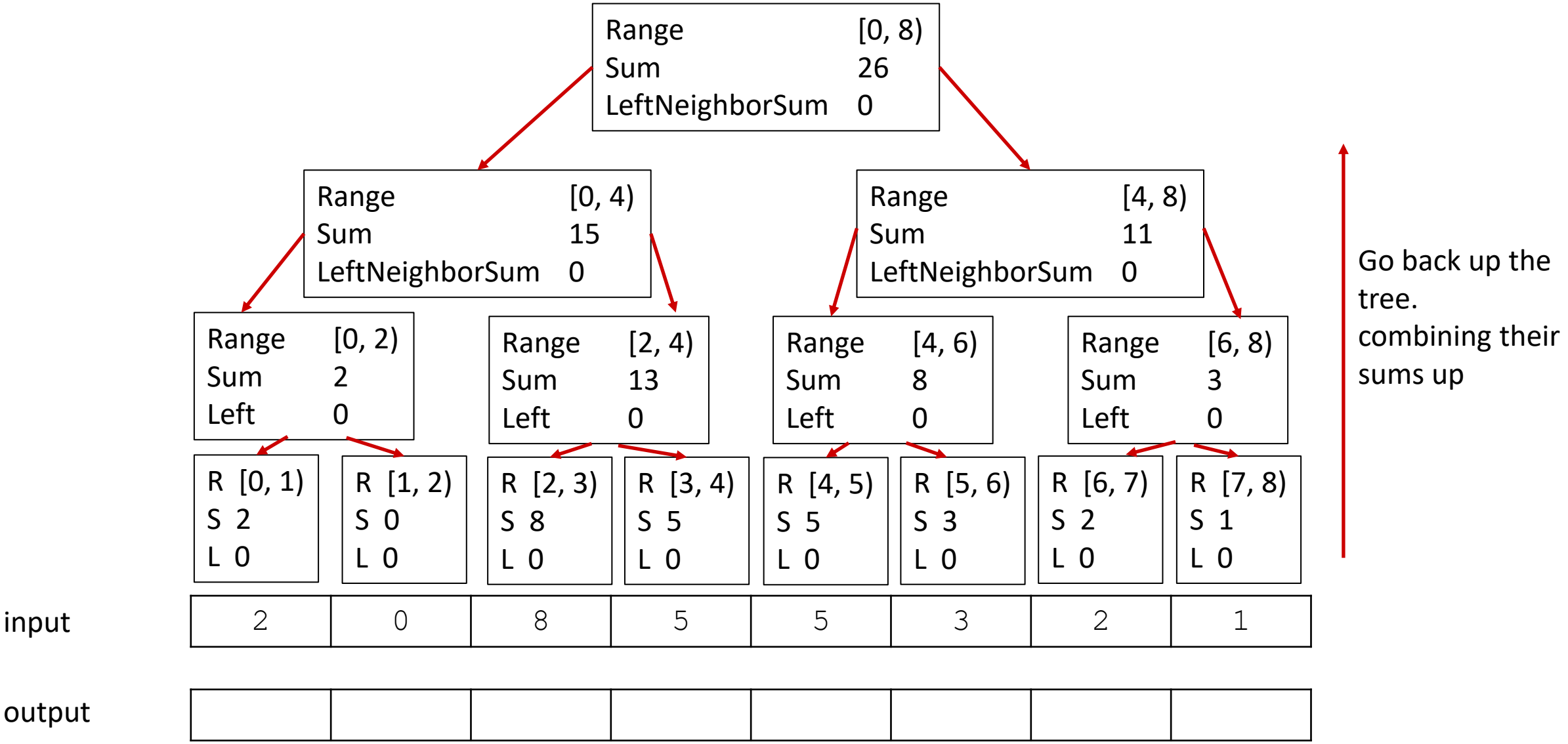
Parallel Prefix Sum

Credit to UW CSE 332 for teaching me this and how to teach this



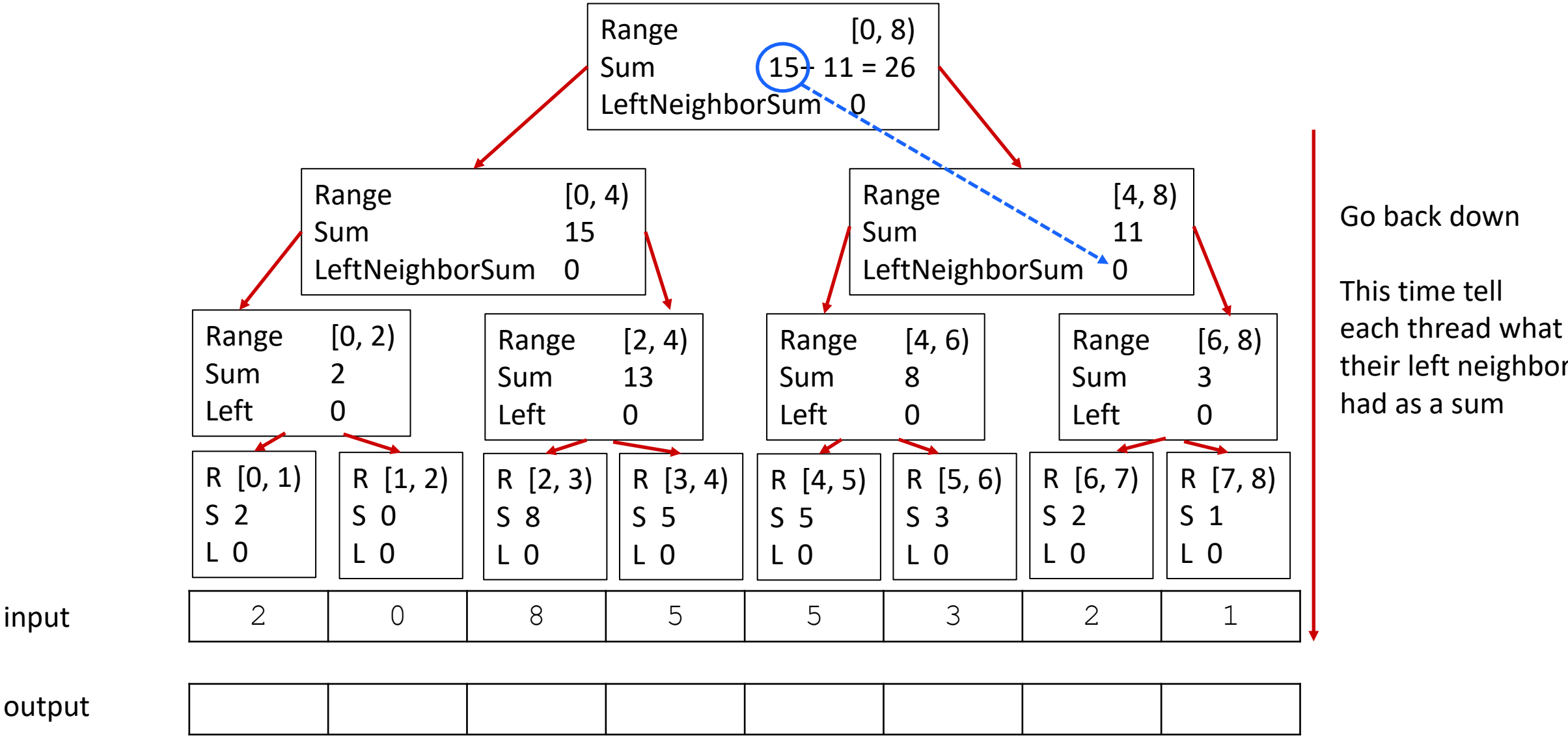
Parallel Prefix Sum

Credit to UW CSE 332 for teaching me this and how to teach this



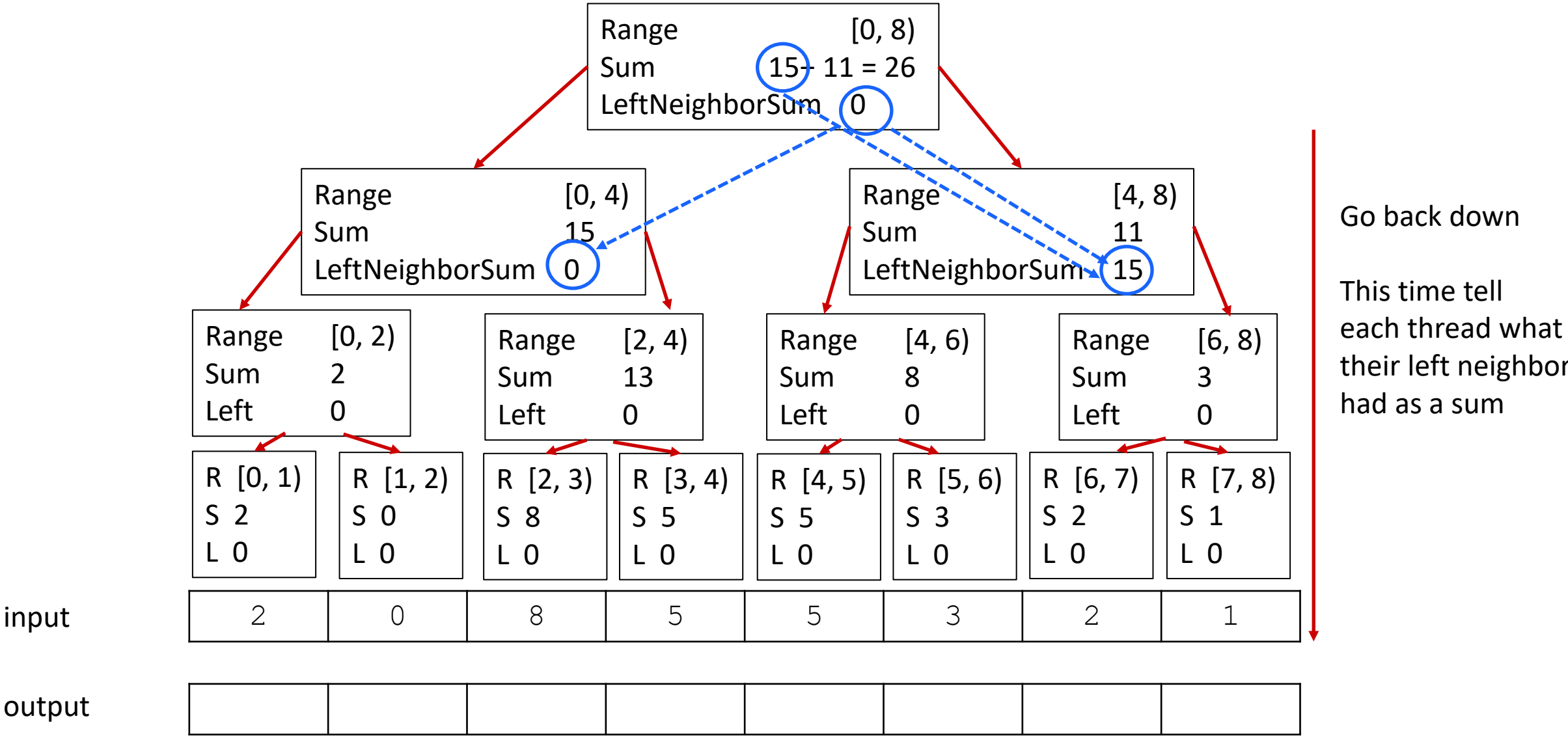
Parallel Prefix Sum

Credit to UW CSE 332 for teaching me this and how to teach this



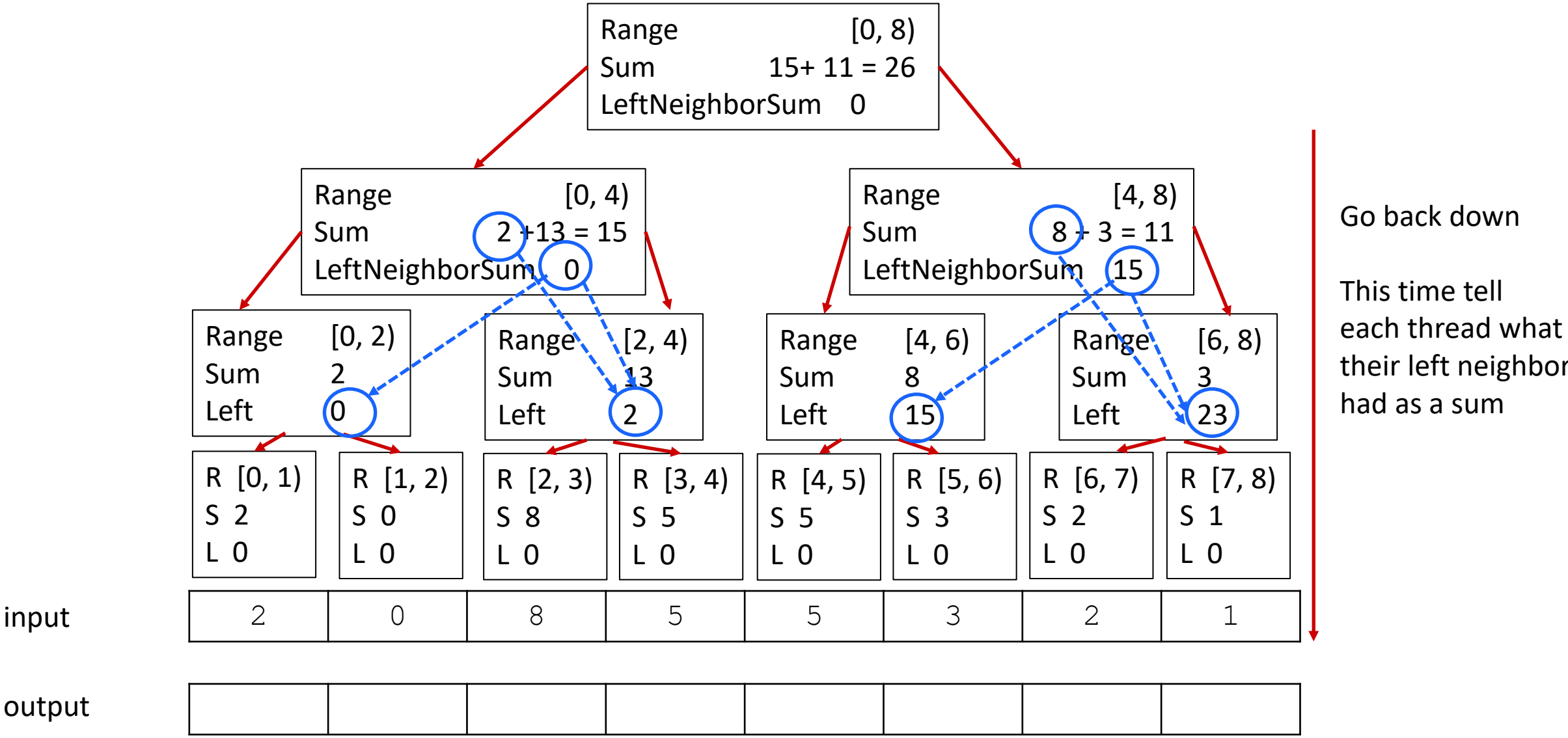
Parallel Prefix Sum

Credit to UW CSE 332 for teaching me this and how to teach this



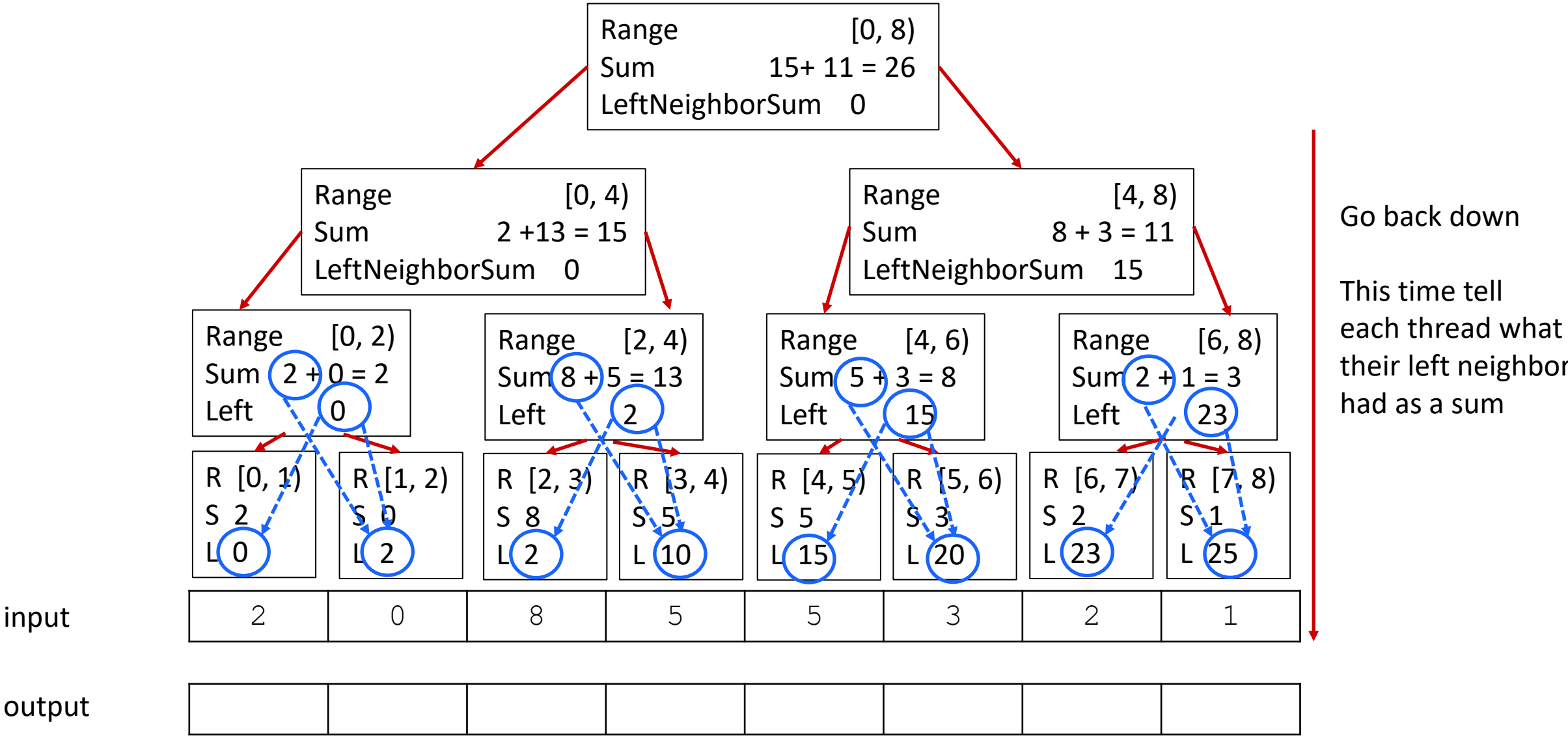
Parallel Prefix Sum

Credit to UW CSE 332 for teaching me this and how to teach this



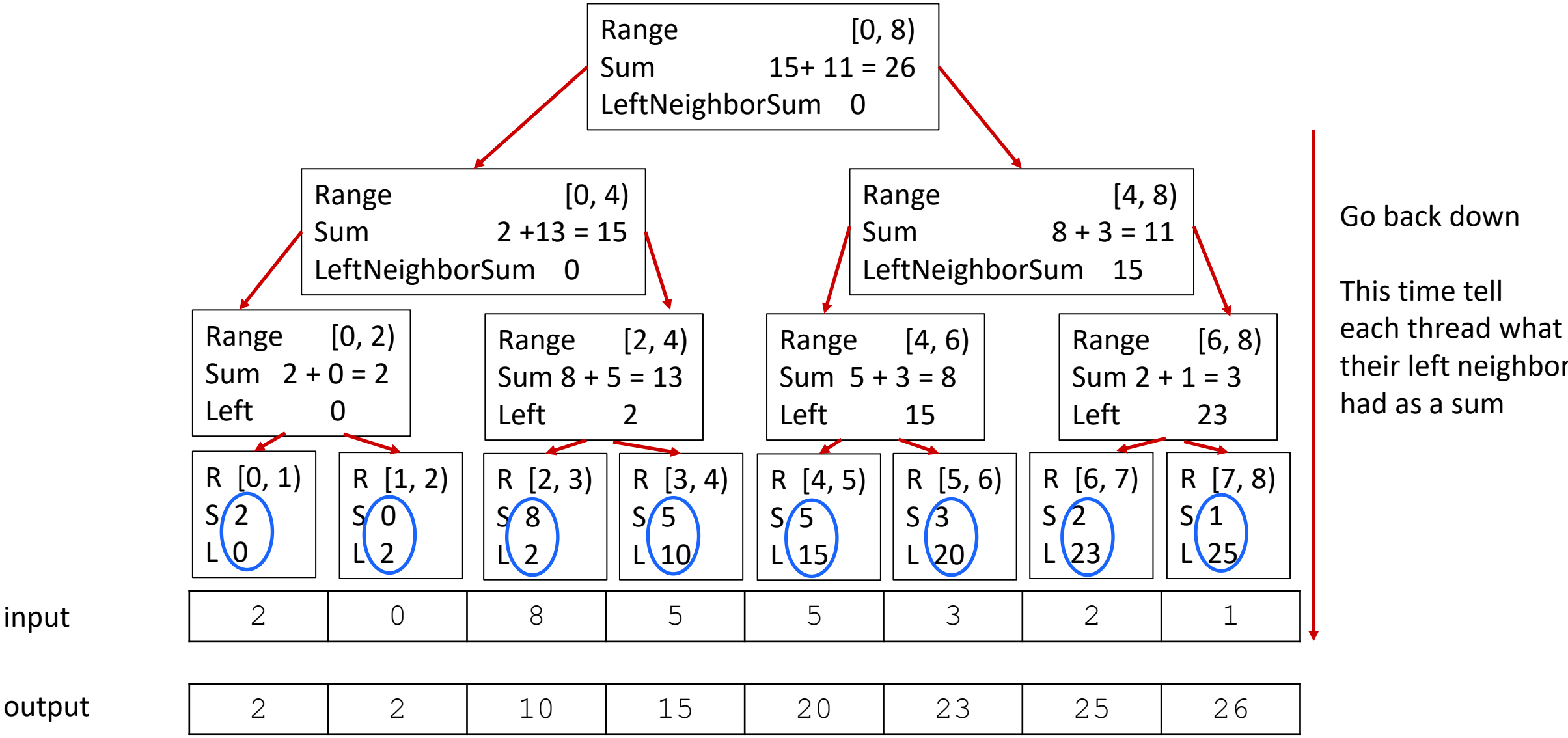
Parallel Prefix Sum

Credit to UW CSE 332 for teaching me this and how to teach this



Parallel Prefix Sum

Credit to UW CSE 332 for teaching me this and how to teach this

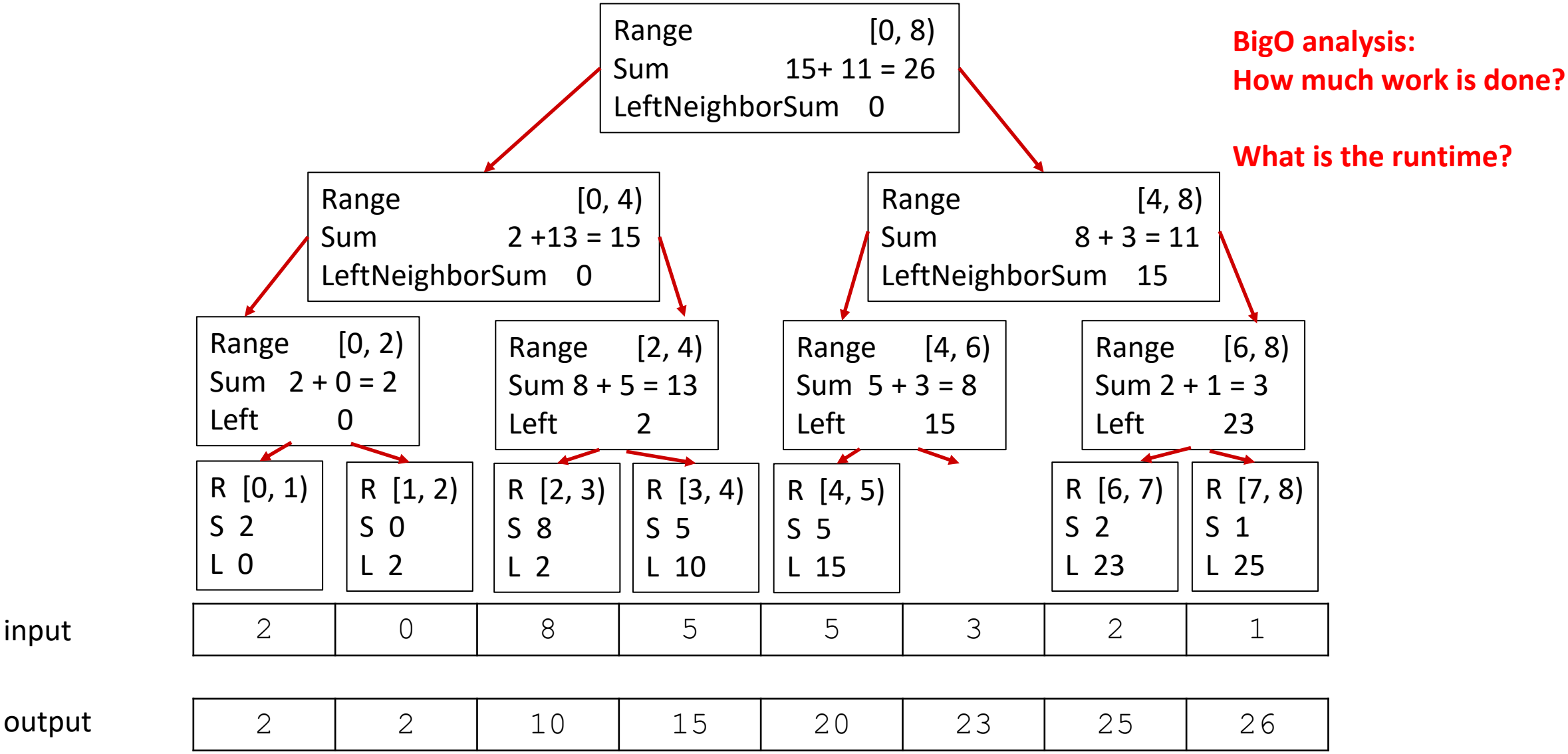


Go back down

This time tell each thread what their left neighbor had as a sum

Parallel Prefix Sum Analysis

Credit to UW CSE 332 for teaching me this and how to teach this



BigO analysis:
How much work is done?

What is the runtime?

Parallel Prefix Sum Analysis

- This algorithm can be tweaked to solve other prefix problems
- What is the minimum/maximum of all elements to the left?
- What is the count of elements to the left that satisfy some property?

Parallel Filter

- Filter higher order function
  - Given a sequence, returns a subsequence containing only elements that meet a certain criteria
- Example:
  - Input vector: `{2, 0, 8, 5, 5, 3, 2, 1}`
  - Function: `auto is_even = [](int n) -> bool { return n % 2 == 0; }`
  - Output vector: `{2, 0, 8, 2}`

- How do we parallelize this (what about with no loops?)
  - Hint: this is made of two parts. *Finding elements that belong in output* and *putting them in the right index in the result.*
  - How do we derive

```
in_output = {1, 1, 1, 0, 0, 0, 1, 0}
indexes   = {0, 1, 2, -, -, -, 3, -}  // - means value doesn't matter
```

Parallel Filter

- Filter higher order function
  - Given a sequence, returns a subsequence containing only elements that meet a certain criteria
- Example:
  - Input vector: `{2, 0, 8, 5, 5, 3, 2, 1}`
  - Function: `auto is_even = [](int n) -> bool { return n % 2 == 0; }`
  - Output vector: `{2, 0, 8, 2}`

- Two* steps:
  - Parallel map/transform to get `in_output = {1, 1, 1, 0, 0, 0, 1, 0}`
  - Parallel prefix sum on previous `indexes = {0, 1, 2, 2, 2, 2, 3, 3}`
    - Right after each thread calculates prefix-sum, have them do --->

```
if (in_output[i] == 1)
   output[indexes[i]] = input[i]
```

That's all for now!

- Next Week:
  - Intro to the Network & Socket programming

- Hopefully you are doing well ☺