Worksheet: Parallelizing the Split step in Radix Sort

The Radix Sort algorithm loops over the bits in the binary representation of the keys, starting at the lowest bit, and executes a split operation for each bit as shown below. The split operation packs the keys with a 0 in the corresponding bit to the front of a vector, and packs the keys with a 1 to the end of the same vector. It maintains the order within both groups.

The sort works because each split operation sorts the keys with respect to the current bit and maintains the sorted order of all the lower bits. Your task is to show how the split operation (complete I-down) can be performed in parallel

```
                                [101 111 011 001 100 010 111 010]
1.A =                           [5 7 3 1 4 2 7 2]
2.A⟨0⟩ =                        [1 1 1 1 0 0 1 0] //lowest bit
3.A←split(A,A⟨0⟩) =             [4 2 2 5 7 3 1 7]
4.A⟨1⟩ =                        [0 1 1 0 1 1 0 1] // middle bit
5.A←split(A,A⟨1⟩) =             [4 5 1 2 2 7 3 7]
6.A⟨2⟩ =                        [1 1 0 0 0 1 0 1] // highest bit
7.A←split(A,A⟨2⟩) =             [1 2 2 3 4 5 7 7]
```

```
procedure split(A, Flags)
  I-down ←
  I-up   ← rev(n - i-scan(+, rev(Flags)) // rev = reverse
  in parallel for each index i
    if (Flags[i])
      Index[i] ← I-up[i]
    else
      Index[i] ← I-down[i]
  result ← permute(A, Index)
```

A	=	[5	7	3	1	4	2	7	2]
Flags	=	[1	1	1	1	0	0	1	0]
I-down	=	[0	0	0	0	0	1	2	2]
I-up	=	[3	4	5	6	6	6	7	7]
Index	=	[3	4	5	6	0	1	7	2]
permute(A, Index)	=	[4	2	2	5	7	3	1	7]

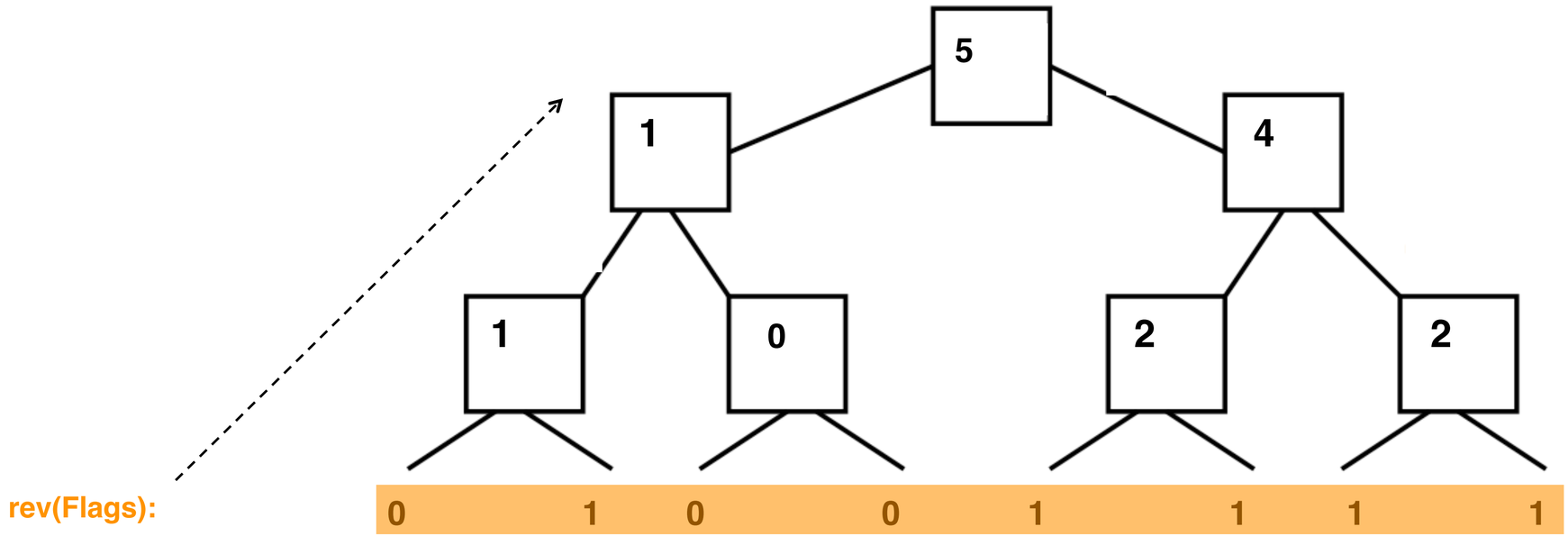
FIGURE 1.9
The split operation packs the elements with a 0 in the corresponding flag position to the bottom of a vector, and packs the elements with a 1 to the top of the same vector. The permute writes each element of A to the index specified by the corresponding position in Index.

Parallel Pre-scan Sum (I-Up): Upward Sweep

Upward sweep is just like Parallel Reduction, except that partial sums are also stored along the way

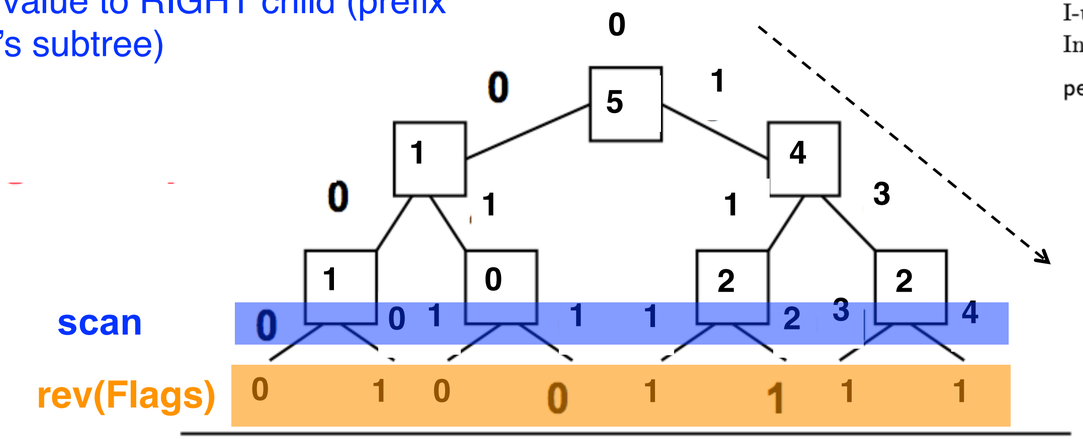
1. Receive values from left and right children
2. Compute left+right and store in box
3. Send left+right value to parent

A	=	[5	7	3	1	4	2	7	2]
Flags	=	[1	1	1	1	0	0	1	0]
I-down	=	[0	0	0	0	0	1	2	2]
I-up	=	[3	4	5	6	6	6	7	7]
Index	=	[3	4	5	6	0	1	7	2]
permute(A, Index)	=	[4	2	2	5	7	3	1	7]



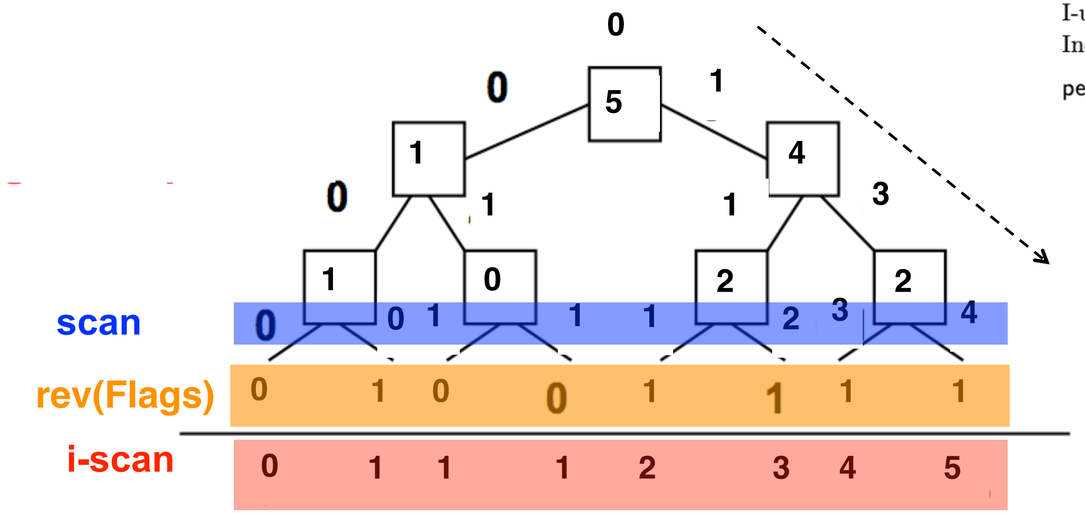
Parallel Scan Sum (I-Up): Downward Sweep

1. Receive value from parent (root receives 0)
2. Send parent's value to LEFT child (prefix sum for elements to left of left child's subtree)
3. Send parent's value+ left child's box value to RIGHT child (prefix sum for elements to left of right child's subtree)
4. Add A[i] to get inclusive prefix sum



A	=	[	5	7	3	1	4	2	7	2]
Flags	=	[	1	1	1	1	0	0	1	0]
I-down	=	[	0	0	0	0	0	1	2	2]
I-up	=	[	3	4	5	6	6	6	7	7]
Index	=	[	3	4	5	6	0	1	7	2]
permute(A, Index)	=	[	4	2	2	5	7	3	1	7]

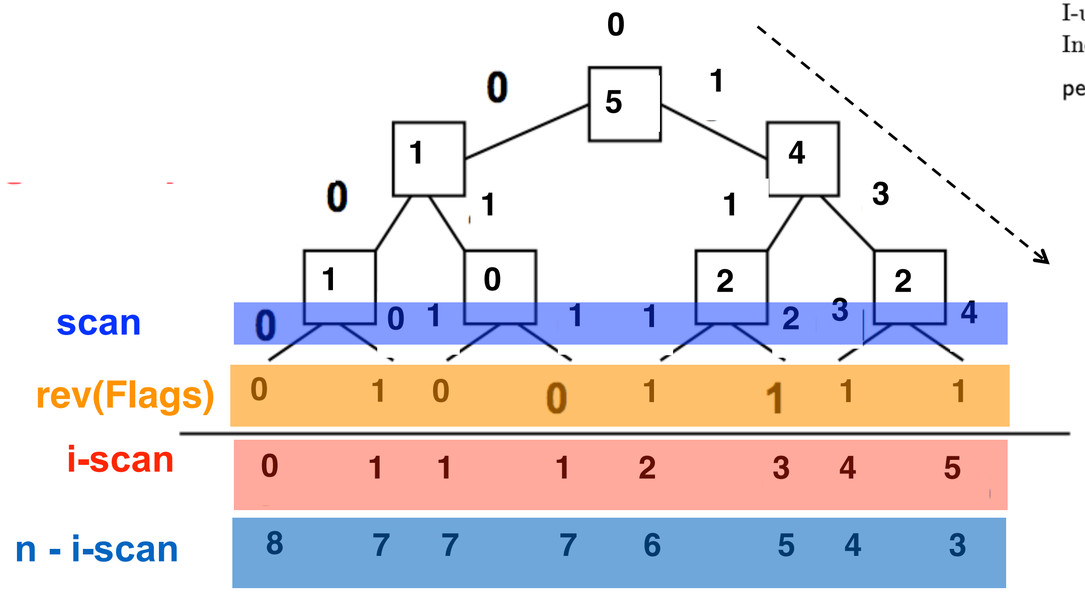
Parallel Scan Sum (I-Up): Downward Sweep



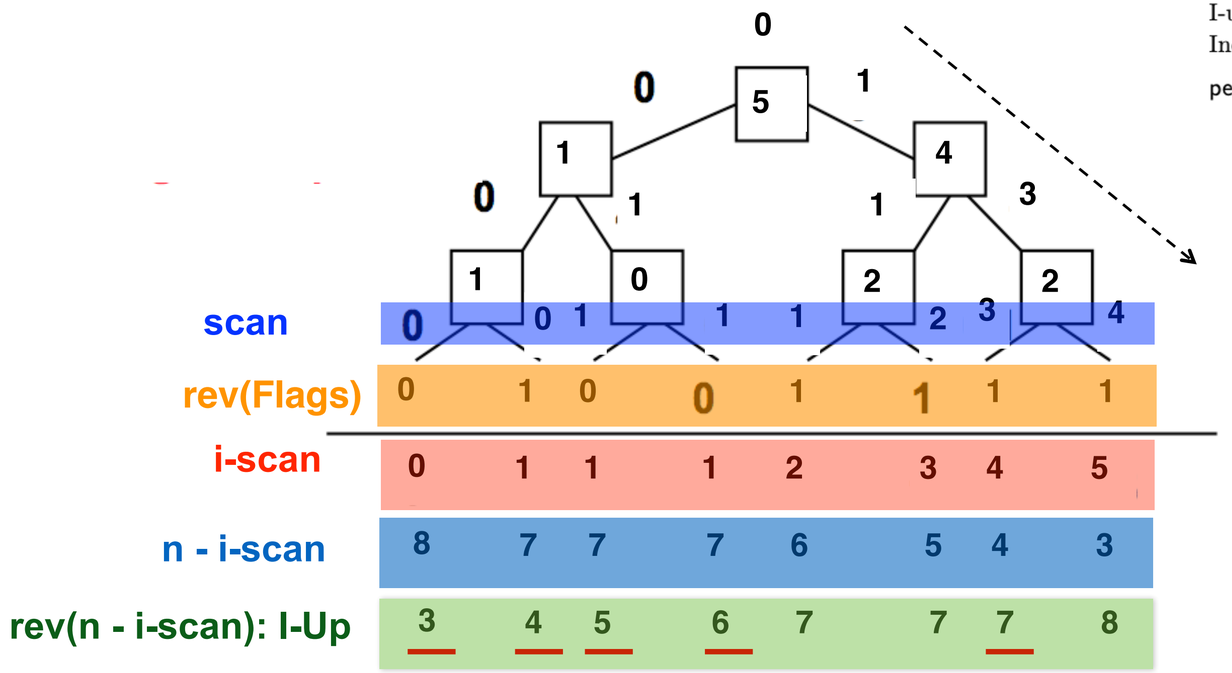
A	=	[	5	7	3	1	4	2	7	2]
Flags	=	[	1	1	1	1	0	0	1	0]
I-down	=	[	0	0	0	0	0	1	2	2]
I-up	=	[	3	4	5	6	6	6	7	7]
Index	=	[	3	4	5	6	0	1	7	2]
permute(A, Index)	=	[	4	2	2	5	7	3	1	7]

Parallel Scan Sum (I-Up): Downward Sweep

A	=	[	5	7	3	1	4	2	7	2]
Flags	=	[	1	1	1	1	0	0	1	0]
I-down	=	[	0	0	0	0	[0]	[1]	2	[2]]
I-up	=	[	[3]	[4]	[5]	[6]	6	6	[7]	7]
Index	=	[	3	4	5	6	0	1	7	2]
permute(A, Index)	=	[	4	2	2	5	7	3	1	7]



Parallel Scan Sum (I-Up): Downward Sweep



A	=	[	5	7	3	1	4	2	7	2]
Flags	=	[	1	1	1	1	0	0	1	0]
I-down	=	[	0	0	0	0	0	1	2	2]
I-up	=	[	3	4	5	6	6	6	7	7]
Index	=	[	3	4	5	6	0	1	7	2]
permute(A, Index)	=	[	4	2	2	5	7	3	1	7]

```
procedure split(A. Flags)
  I-down ← .....
  I-up   ← rev(n - i-scan(+, rev(Flags)) // rev = reverse
  in parallel for each index i
    if (Flags[i])
      Index[i] ← I-up[i]
    else
      Index[i] ← I-down[i]
  result ← permute(A, Index)
```

Worksheet: Parallelizing the Split step in Radix Sort

The Radix Sort algorithm loops over the bits in the binary representation of the keys, starting at the lowest bit, and executes a split operation for each bit as shown below. The split operation packs the keys with a 0 in the corresponding bit to the front of a vector, and packs the keys with a 1 to the end of the same vector. It maintains the order within both groups.

The sort works because each split operation sorts the keys with respect to the current bit and maintains the sorted order of all the lower bits. Your task is to show how the split operation (complete I-down) can be performed in parallel

```
                                [101 111 011 001 100 010 111 010]
1.A =                           [5 7 3 1 4 2 7 2]
2.A⟨0⟩ =                        [1 1 1 1 0 0 1 0] //lowest bit
3.A←split(A,A⟨0⟩) =             [4 2 2 5 7 3 1 7]
4.A⟨1⟩ =                        [0 1 1 0 1 1 0 1] // middle bit
5.A←split(A,A⟨1⟩) =             [4 5 1 2 2 7 3 7]
6.A⟨2⟩ =                        [1 1 0 0 0 1 0 1] // highest bit
7.A←split(A,A⟨2⟩) =             [1 2 2 3 4 5 7 7]
```

```
procedure split(A, Flags)
  I-down ←
  I-up   ← rev(n - i-scan(+, rev(Flags)) // rev = reverse
  in parallel for each index i
    if (Flags[i])
      Index[i] ← I-up[i]
    else
      Index[i] ← I-down[i]
  result ← permute(A, Index)
```

A	=	[5	7	3	1	4	2	7	2]
Flags	=	[1	1	1	1	0	0	1	0]
I-down	=	[0	0	0	0	0	1	2	2]
I-up	=	[3	4	5	6	6	6	7	7]
Index	=	[3	4	5	6	0	1	7	2]
permute(A, Index)	=	[4	2	2	5	7	3	1	7]

FIGURE 1.9
The split operation packs the elements with a 0 in the corresponding flag position to the bottom of a vector, and packs the elements with a 1 to the top of the same vector. The permute writes each element of A to the index specified by the corresponding position in Index.

Worksheet: Parallelizing the Split step in Radix Sort

The Radix Sort algorithm loops over the bits in the binary representation of the keys, starting at the lowest bit, and executes a split operation for each bit as shown below. The split operation packs the keys with a 0 in the corresponding bit to the front of a vector, and packs the keys with a 1 to the end of the same vector. It maintains the order within both groups.

The sort works because each split operation sorts the keys with respect to the current bit and maintains the sorted order of all the lower bits. Your task is to show how the split operation (complete I-down) can be performed in parallel

```
                                [101 111 011 001 100 010 111 010]
1.A =                           [5 7 3 1 4 2 7 2]
2.A⟨0⟩ =                        [1 1 1 1 0 0 1 0] //lowest bit
3.A←split(A,A⟨0⟩) =             [4 2 2 5 7 3 1 7]
4.A⟨1⟩ =                        [0 1 1 0 1 1 0 1] // middle bit
5.A←split(A,A⟨1⟩) =             [4 5 1 2 2 7 3 7]
6.A⟨2⟩ =                        [1 1 0 0 0 1 0 1] // highest bit
7.A←split(A,A⟨2⟩) =             [1 2 2 3 4 5 7 7]
```

```
procedure split(A, Flags)
  I-down ← scan(+, not(Flags))
  I-up   ← rev(n - i-scan(+, rev(Flags)) // rev = reverse
  in parallel for each index i
    if (Flags[i])
      Index[i] ← I-up[i]
    else
      Index[i] ← I-down[i]
  result ← permute(A, Index)
```

A	=	[5	7	3	1	4	2	7	2]
Flags	=	[1	1	1	1	0	0	1	0]
I-down	=	[0	0	0	0	0	1	2	2]
I-up	=	[3	4	5	6	6	6	7	7]
Index	=	[3	4	5	6	0	1	7	2]
permute(A, Index)	=	[4	2	2	5	7	3	1	7]

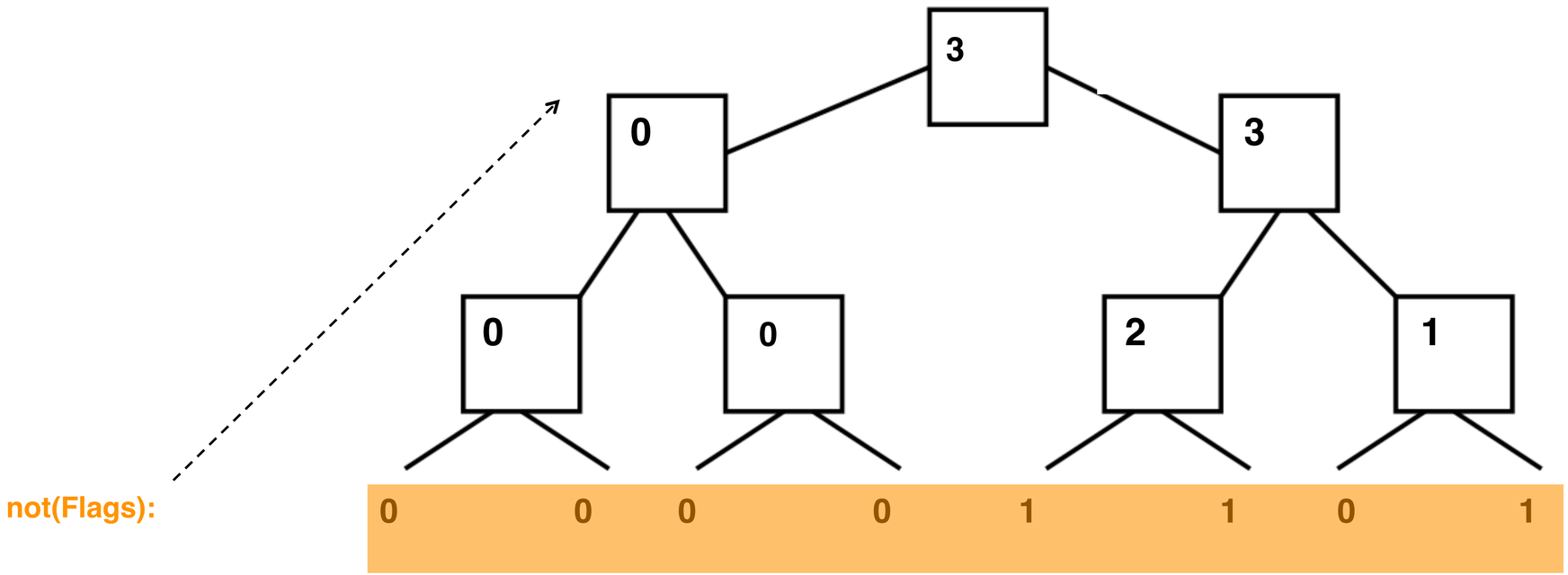
FIGURE 1.9
The split operation packs the elements with a 0 in the corresponding flag position to the bottom of a vector, and packs the elements with a 1 to the top of the same vector. The permute writes each element of A to the index specified by the corresponding position in Index.

Parallel Scan Sum (I-Down): Upward Sweep

Upward sweep is just like Parallel Reduction, except that partial sums are also stored along the way

1. Receive values from left and right children
2. Compute left+right and store in box
3. Send left+right value to parent

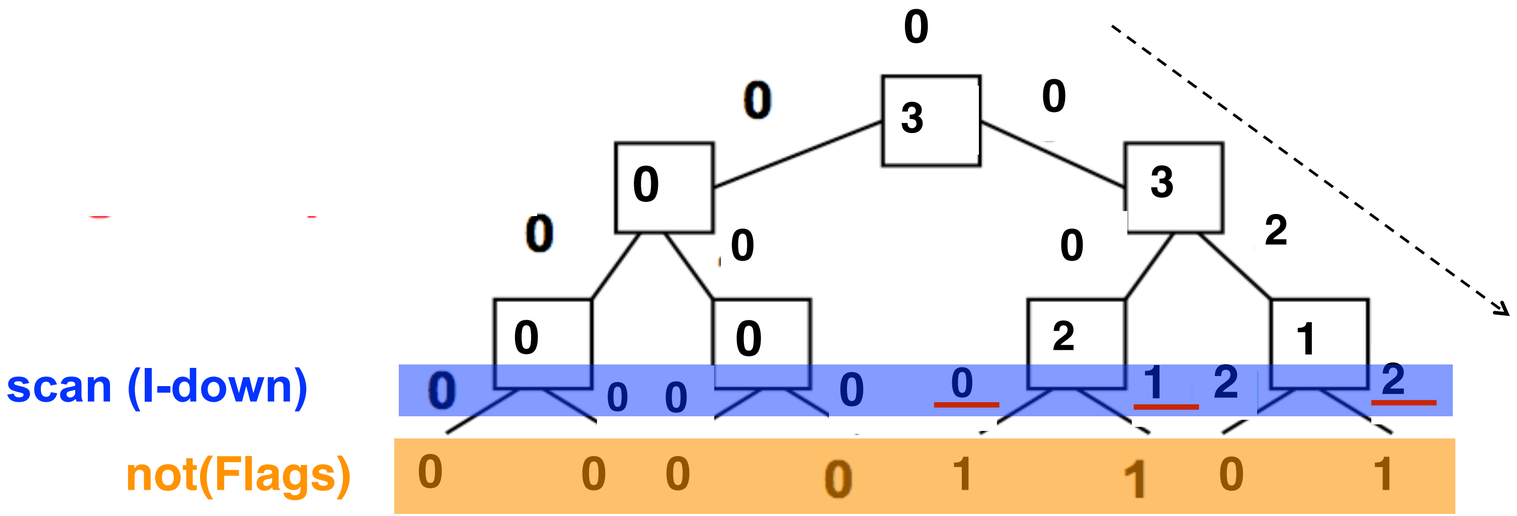
A	=	[5	7	3	1	4	2	7	2]
Flags	=	[1	1	1	1	0	0	1	0]
I-down	=	[0	0	0	0	[0]	[1]	2	[2]]
I-up	=	[[3]	[4]	[5]	[6]	6	6	[7]	7]
Index	=	[3	4	5	6	0	1	7	2]
permute(A, Index)	=	[4	2	2	5	7	3	1	7]



Parallel Scan Sum (I-Down): Downward Sweep

1. Receive value from parent (root receives 0)
2. Send parent's value to LEFT child (prefix sum for elements to left of left child's subtree)
3. Send parent's value+ left child's box value to RIGHT child (prefix sum for elements to left of right child's subtree)
4. Add A[i] to get inclusive prefix sum

A	=	[5	7	3	1	4	2	7	2]
Flags	=	[1	1	1	1	0	0	1	0]
I-down	=	[0	0	0	0	[0]	[1]	2	[2]]
I-up	=	[[3]	[4]	[5]	[6]	6	6	[7]	7]
Index	=	[3	4	5	6	0	1	7	2]
permute(A, Index)	=	[4	2	2	5	7	3	1	7]




```
procedure split(A. Flags)
  I-down ← .....
  I-up   ← rev(n - i-scan(+, rev(Flags)) // rev = reverse
  in parallel for each index i
    if (Flags[i])
      Index[i] ← I-up[i]
    else
      Index[i] ← I-down[i]
  result ← permute(A, Index)
```

Worksheet: Parallelizing the Split step in Radix Sort

The Radix Sort algorithm loops over the bits in the binary representation of the keys, starting at the lowest bit, and executes a split operation for each bit as shown below. The split operation packs the keys with a 0 in the corresponding bit to the front of a vector, and packs the keys with a 1 to the end of the same vector. It maintains the order within both groups.

The sort works because each split operation sorts the keys with respect to the current bit and maintains the sorted order of all the lower bits. Your task is to show how the split operation (complete I-down) can be performed in parallel

```
                                [101 111 011 001 100 010 111 010]
1.A =                           [5 7 3 1 4 2 7 2]
2.A⟨0⟩ =                        [1 1 1 1 0 0 1 0] //lowest bit
3.A←split(A,A⟨0⟩) =             [4 2 2 5 7 3 1 7]
4.A⟨1⟩ =                        [0 1 1 0 1 1 0 1] // middle bit
5.A←split(A,A⟨1⟩) =             [4 5 1 2 2 7 3 7]
6.A⟨2⟩ =                        [1 1 0 0 0 1 0 1] // highest bit
7.A←split(A,A⟨2⟩) =             [1 2 2 3 4 5 7 7]
```

```
procedure split(A, Flags)
  I-down ← scan(+, not(Flags))
  I-up   ← rev(n - i-scan(+, rev(Flags)) // rev = reverse
  in parallel for each index i
    if (Flags[i])
      Index[i] ← I-up[i]
    else
      Index[i] ← I-down[i]
  result ← permute(A, Index)
```

A	=	[5	7	3	1	4	2	7	2]
Flags	=	[1	1	1	1	0	0	1	0]
I-down	=	[0	0	0	0	0	1	2	2]
I-up	=	[3	4	5	6	6	6	7	7]
Index	=	[3	4	5	6	0	1	7	2]
permute(A, Index)	=	[4	2	2	5	7	3	1	7]

FIGURE 1.9
The split operation packs the elements with a 0 in the corresponding flag position to the bottom of a vector, and packs the elements with a 1 to the top of the same vector. The permute writes each element of A to the index specified by the corresponding position in Index.

Another way of looking at the parallel algorithm

Observation: each prefix sum can be decomposed into reusable terms of power-of-2-size e.g.

$$\begin{aligned} X[6] &= A[0] + A[1] + A[2] + A[3] + A[4] + A[5] + A[6] \\ &= (A[0] + A[1] + A[2] + A[3]) + (A[4] + A[5]) + A[6] \end{aligned}$$

Approach:

- Combine reduction tree idea from Parallel Array Sum with partial sum idea from Sequential Prefix Sum
- Use an “upward sweep” to perform parallel reduction, while storing partial sum terms in tree nodes
- Use a “downward sweep” to compute prefix sums while reusing partial sum terms stored in upward sweep