Week 7 Solutions

CSE 332 Summer 2015

Quickcheck and Group Problems

Quickcheck) Parallel Prefix Sum

Goal: Output array needs to store sums of everything up to a certain index. Meaning:
Output[i] = input[i]+input[i-1]+input[i-2]+…+input[0]

input	8	9	6	3	2	5	7	2
output								

Figure out what information you need

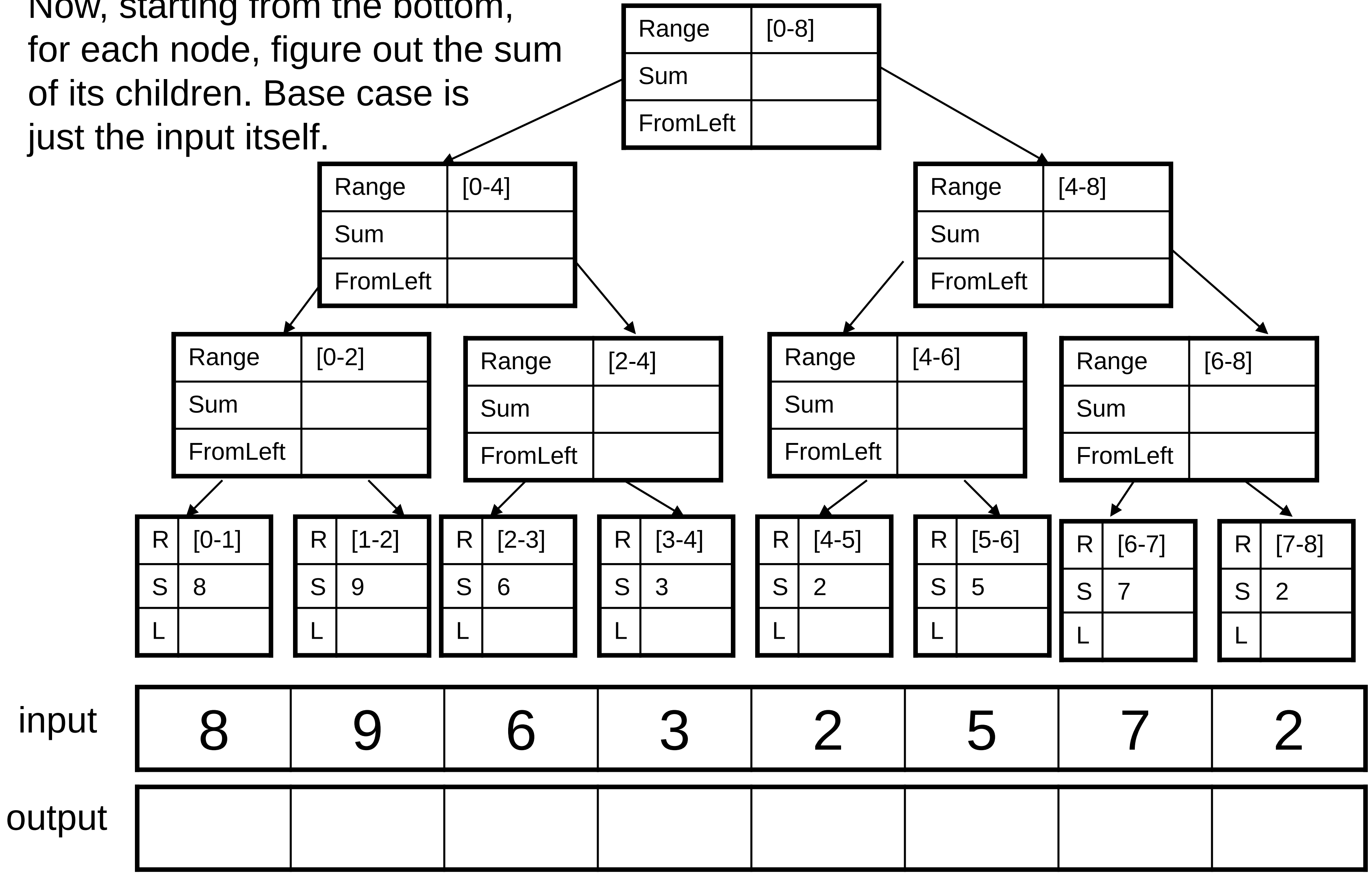
Range	[0-8]
Sum	
FromLeft	

Start off at root with the entire range of the problem (low=0, high=8). We need to find the Sum and the FromLeft value of the root, but we will do this in two passes. First pass, go down and split up the problem until we get to the cutoff of one item (high-low=1)

input	8	9	6	3	2	5	7	2
output								

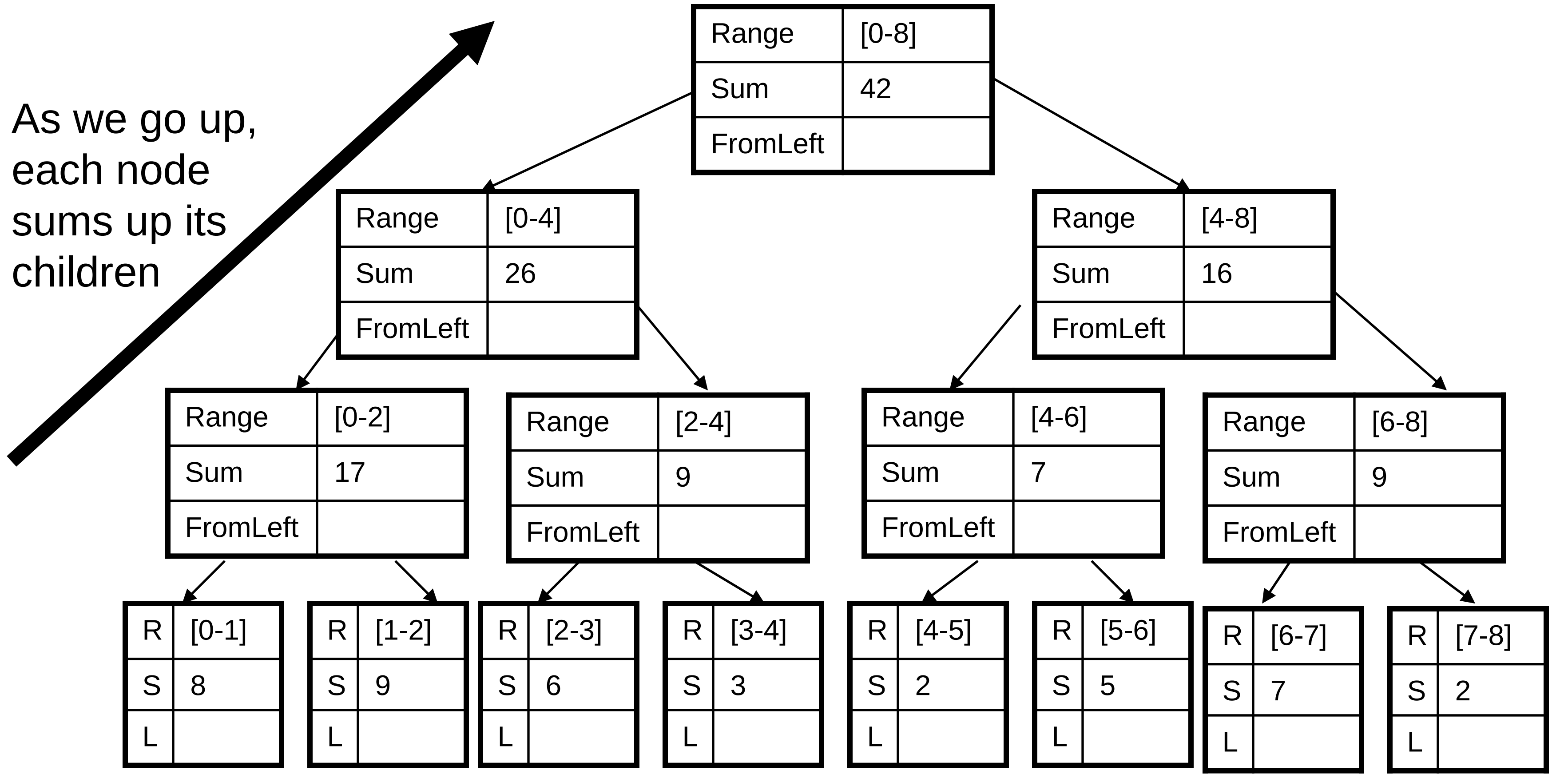
Divide problem into parallel pieces

Now, starting from the bottom, for each node, figure out the sum of its children. Base case is just the input itself.



1st pass, find sums going up.

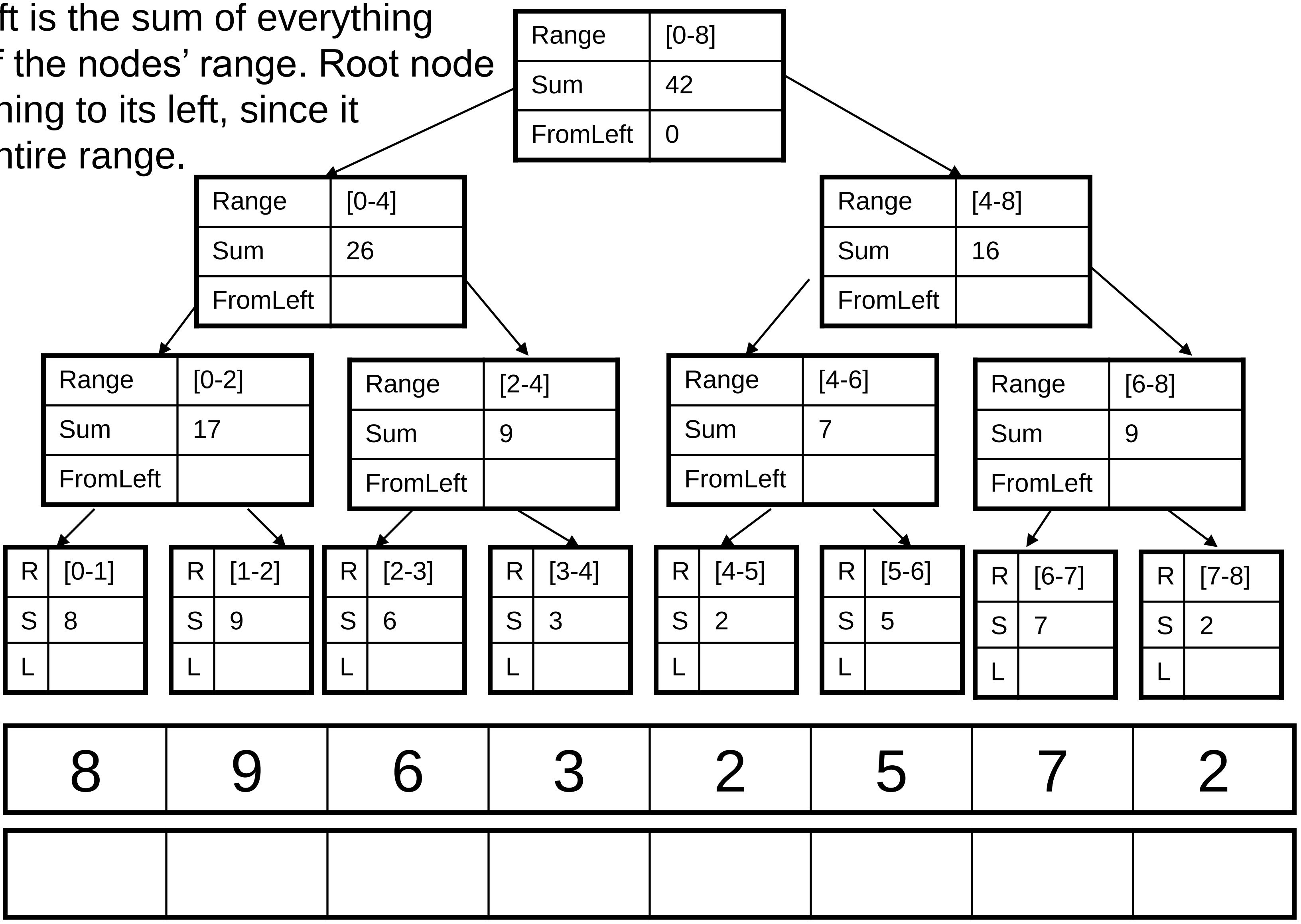
As we go up, each node sums up its children



input	8	9	6	3	2	5	7	2
output								

2nd pass, fill out FromLeft going down

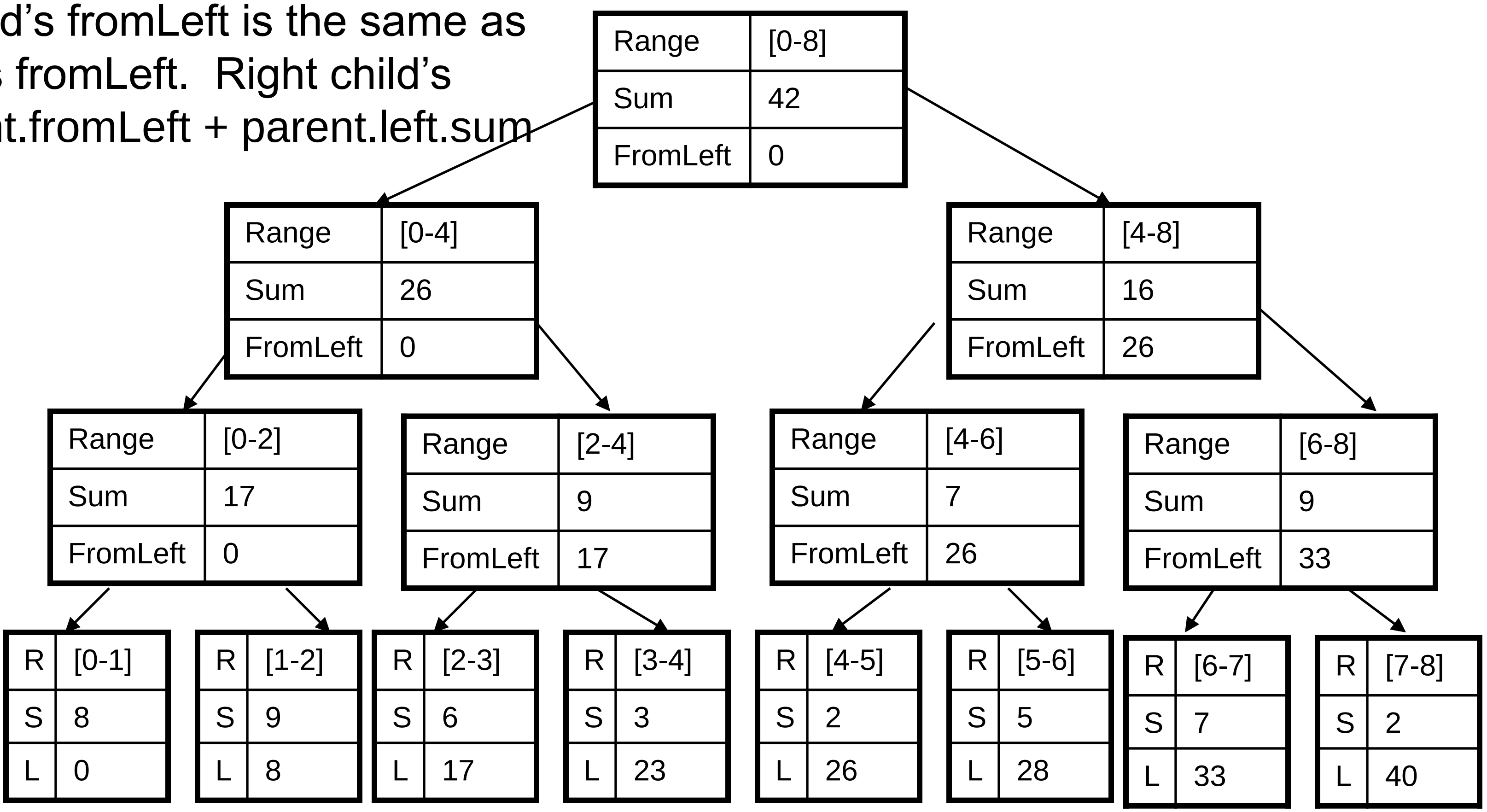
From left is the sum of everything LEFT of the nodes' range. Root node has nothing to its left, since it Is the entire range.



input	8	9	6	3	2	5	7	2
output								

2nd pass, fill out FromLeft going down

Left child's fromLeft is the same as parent's fromLeft. Right child's is parent.fromLeft + parent.left.sum

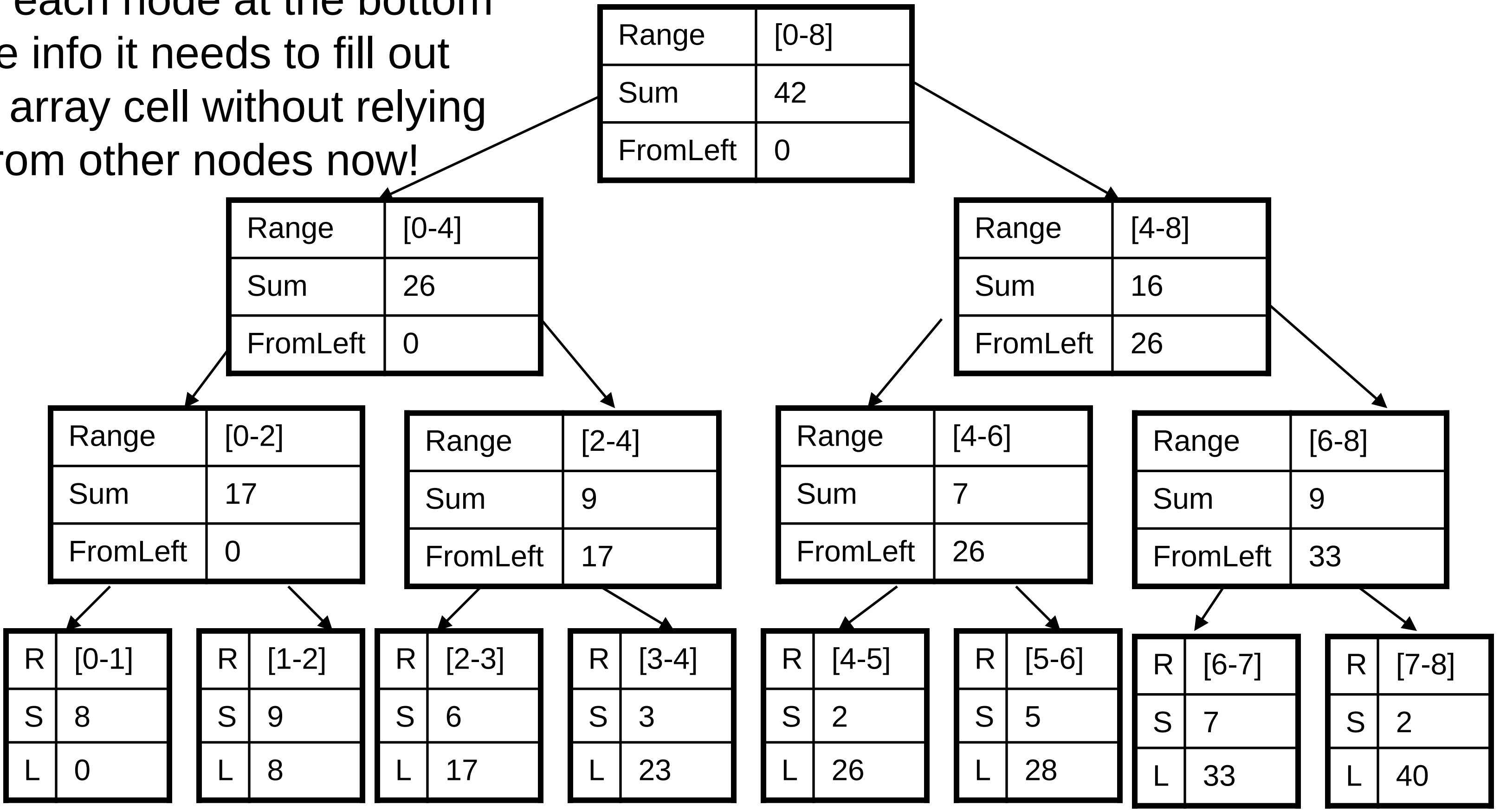


input	8	9	6	3	2	5	7	2
output								

Finally, fill out output array

output[this.low] = this.sum + this.fromLeft

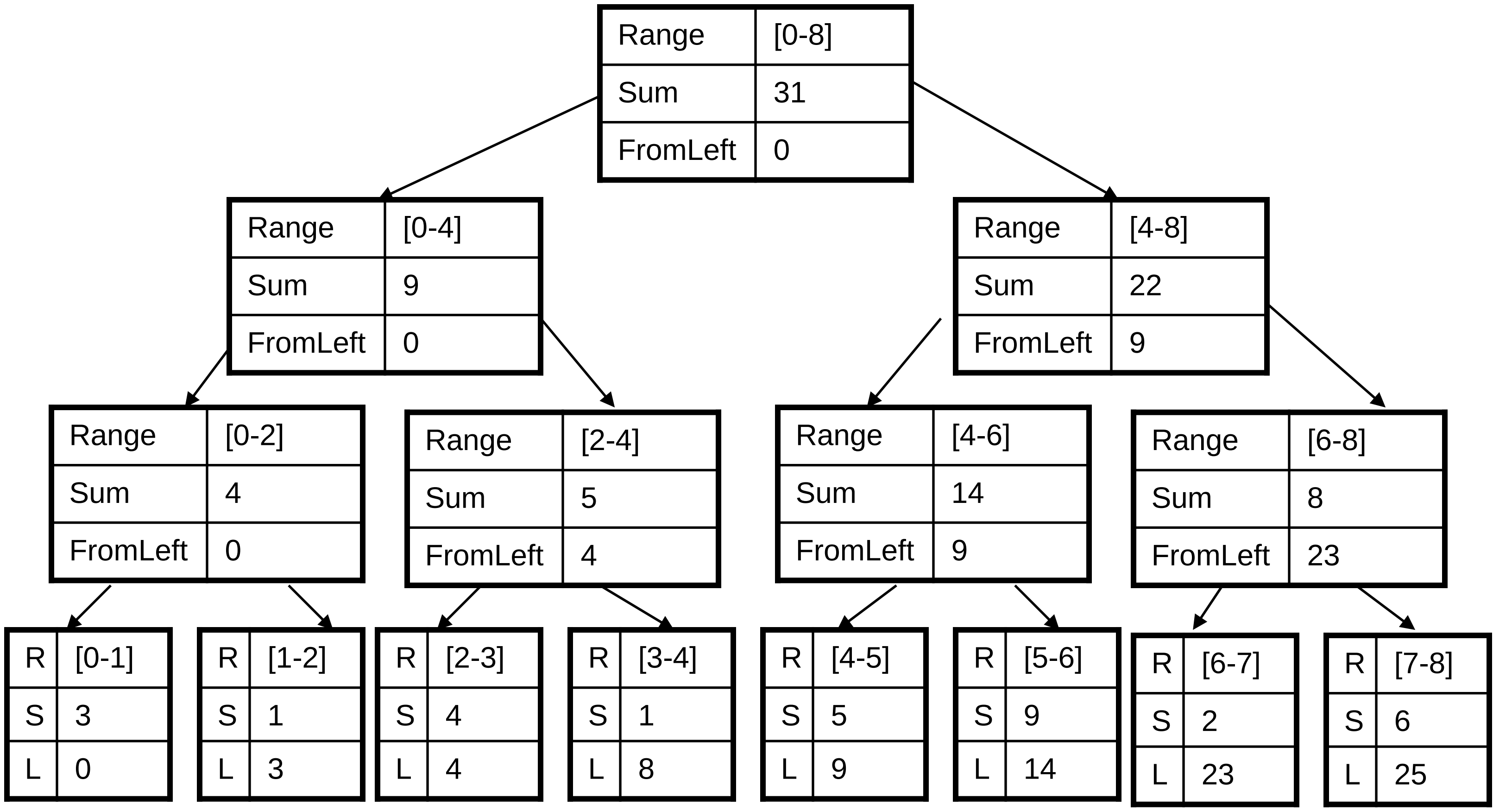
Basically, each node at the bottom has all the info it needs to fill out its output array cell without relying on data from other nodes now!


Range [0-8] Sum 42 FromLeft 0
Range [0-4] Sum 26 FromLeft 0
Range [4-8] Sum 16 FromLeft 26
Range [0-2] Sum 17 FromLeft 0
Range [2-4] Sum 9 FromLeft 17
Range [4-6] Sum 7 FromLeft 26
Range [6-8] Sum 9 FromLeft 33
R [0-1] S 8 L 0
R [1-2] S 9 L 8
R [2-3] S 6 L 17
R [3-4] S 3 L 23
R [4-5] S 2 L 26
R [5-6] S 5 L 28
R [6-7] S 7 L 33
R [7-8] S 2 L 40

input	8	9	6	3	2	5	7	2
output	8	17	23	26	28	33	40	42

Now for the group problems:

1: Prefix Sum of pi



input	3	1	4	1	5	9	2	6
output	8	17	23	26	28	33	40	42

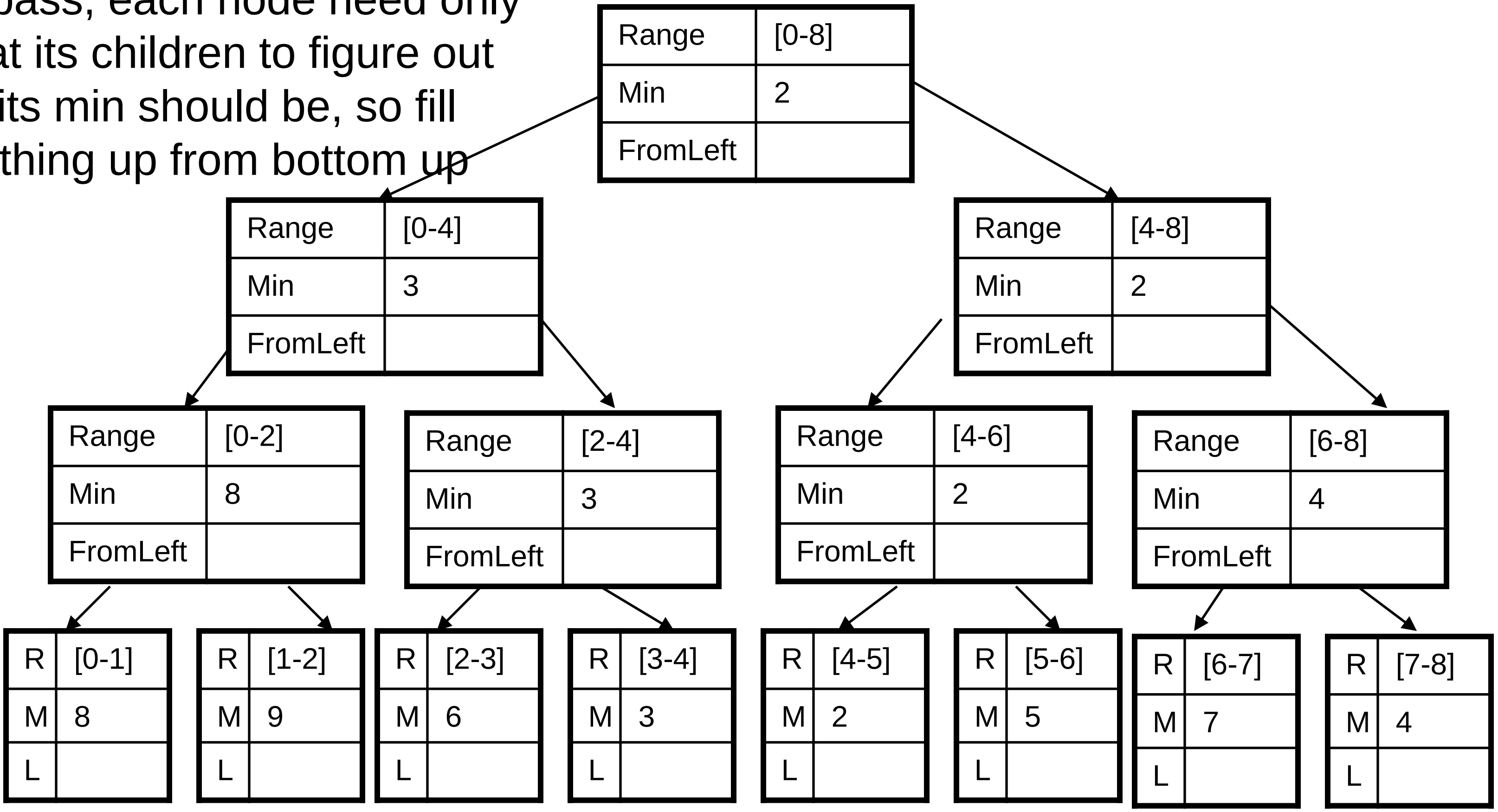
2) Parallel Prefix FindMin

Output an array with the minimum value of all cells to its left.
So, output[i] = min(input[0],input[1],input[2],….input[i-1],input[i])

input	8	9	6	3	2	5	7	4
output								

Same as before, except this time, we want to store the node's range, the min of its children, and the min of everything to its left.

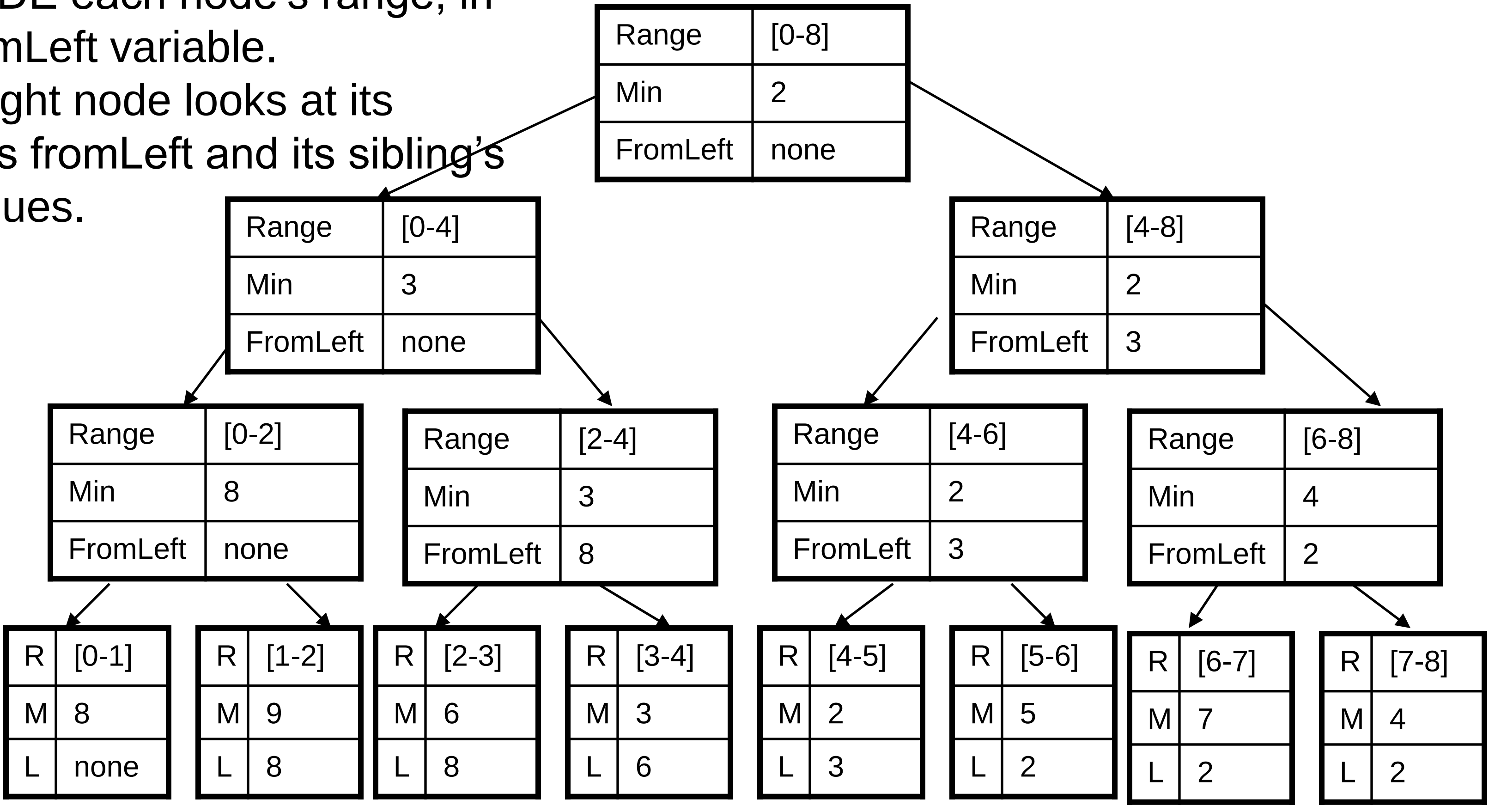
First pass, each node need only look at its children to figure out what its min should be, so fill everything up from bottom up


Range [0-8] Min 2 FromLeft
Range [0-4] Min 3 FromLeft
Range [4-8] Min 2 FromLeft
Range [0-2] Min 8 FromLeft
Range [2-4] Min 3 FromLeft
Range [4-6] Min 2 FromLeft
Range [6-8] Min 4 FromLeft
R [0-1] M 8 L
R [1-2] M 9 L
R [2-3] M 6 L
R [3-4] M 3 L
R [4-5] M 2 L
R [5-6] M 5 L
R [6-7] M 7 L
R [7-8] M 4 L

input	8	9	6	3	2	5	7	4
output								

Second pass, we need to fill everything starting from the root going down.

Fill out the min value from OUTSIDE each node's range, in the fromLeft variable.
Each right node looks at its parent's fromLeft and its sibling's min values.



Range [0-8] Min 2 FromLeft none
Range [0-4] Min 3 FromLeft none
Range [4-8] Min 2 FromLeft 3
Range [0-2] Min 8 FromLeft none
Range [2-4] Min 3 FromLeft 8
Range [4-6] Min 2 FromLeft 3
Range [6-8] Min 4 FromLeft 2
R [0-1] M 8 L none
R [1-2] M 9 L 8
R [2-3] M 6 L 8
R [3-4] M 3 L 6
R [4-5] M 2 L 3
R [5-6] M 5 L 2
R [6-7] M 7 L 2
R [7-8] M 4 L 2

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

3) Quicksort Recurrence Relations

- Recall that sequential Quicksort consists of
  - O(1) Picking a pivot
  - O(n) Partition data into
    - A: Less than pivot
    - B: Pivot
    - C: Greater than pivot
  - 2 T(n/2) – Recursively, sort each of the two halves, A and C.
- $T(n)=1+n+2T(n/2) = O(n \log n)$

To parallelize step 3 (recursion)

- Each partition can be done at the same, so 2T(n/2) becomes time 1 T(n/2)
- Whole relation becomes: T(n)=1+n+T(n/2)
- Ignoring the constant time pivot-picking:
- T(n) = n + T(n/2)

Solve recurrence relation

- $T(n) = n + T(n/2)$
- $T(n) = n + (n/2 + T(n/4))$
- $T(n) = n + (n/2 + (n/4 + T(n/8)))$
- $T(n) = n*(1+1/2+1/4+\ldots+1/2^{k-1})+T(n/2^k)$

Assume T(1)=C, that is, that to sort 1 element takes a constant C units of time.

Substitute in base case T(1)=1 and solve for k:
$n/2^k=1$
$k = \log n$

- $T(n) = n*(1+1/2+1/4+\ldots+1/2^{\log n-1})+C$
- Sum of geometric series (1+1/2+1/4+…) converges to 2
- T(n) = 2n+C which is O(n), linear

4) Parallelizing step 2, partition

- Do 2 filters, one to filter less-than-pivot partition, one to filter greater-than-pivot partition.
- Filter is work O(n), span O(log n)
- So total quicksort is now (partition+recursion):
- T(n) = O(log n) + T(n/2)

Solve recurrence relation

- T(n) = log n + T(n/2) *expand out recurrence*
- T(n) = log n + (log(n/2) + T(n/4))
- T(n) = log n + log(n/2) + log(n/4) + T(n/8)
- T(n) = log n + log(n/2) + log(n/4) + log(n/8) + T(n/16)
- T(n) = log n +(log n – log 2) + (log n – log 4) + (log n – log 8) + T(n/16)
- T(n) = 4*log n – log 2 – log 4 – log 8 + T(n/16)
- T(n) = 4*log n – 1 – 2 – 3 + T(n/2^4) *because we're doing log base 2*
- T(n) = k*log n - (1+2+3+…+(k-1))+T(n/2^k)
- T(n) = k*log n – (k(k-1))/2 + T(n/2^k)
- As usual, assuming T(1)=C, set n/2^k=1, gives k=log n
- T(n) = (log n)*(log n) – ((log n-1)(log n))/2 + C
- T(n) = (log n)*(log n) – ((log n * log n)-log n)/2 + C
- Which is O(log n * log n)

5) Concurrency: Data Races

- Question from CSE332 10su final

```
Consider the code below:
public class StringArray {
      int len=0;
      String[] arr=new String[4];
      synchronized void checkAndResize(){
            //checks to see if array is full; if so, doubles size and copies over
            //if not, returns (nothing changes)
      }
      synchronized int getLen(){return len;}
      synchronized void incLen(){len++;}
      synchronized void set(int index,String s){arr[index]=s;}
      void add(String s){
            checkAndResize();
            int myLength=getLen();
            set(myLength,s);
            incLen();
       }}
```

- Does it have a data race? Explain
  - Yes it has a data race, although methods are individually locked, they share resources.

5b) Interleaving that loses a String

Process 1: add(…)

- checkAndResize();
- int myLength=getLen();
- set(myLength,s);
- incLen();

Process 2: add(…)

- checkAndResize();
- int myLength=getLen();
- set(myLength,s);
- incLen();

5c) Interleaving that throws OOB

Process 1: add(…)

- checkAndResize();

Process 2: add(…)

- checkAndResize();

Process 1:

- int myLength=getLen();
- set(myLength,s);
- incLen();

Process 2:

- int myLength=getLen();
- set(myLength,s);
- incLen();