SEGMENT TREES

ADVANCED ALGORITHMS
MARK FLORYAN

MATERIAL FROM:
HTTPS://CP-ALGORITHMS.COM/DATA_STRUCTURES/SEGMENT_TREE.HTML

ADVANCED TREE STRUCTURES

In this deck we will look at:

- ***Segment Trees***

MOTIVATION: RANGE QUERIES OVER ARRAYS

MOTIVATION

Goal: Given a list of integers $A = \{a_1, a_2, \ldots, a_n\}$ and a function that operates on continuous ranges in A, called $f(l, r)$ where $l, r \in Z; l \leq r$

Support the following operations:

1. Calculate (for any l,r) the value of $f(l, r) = f(a_l, \ldots, a_r)$ in $O(\log n)$ time
2. Update the value of an element of A in $O(\log n)$ time.
3. Use no more than $O(n)$ memory (so no more than list A itself times a constant)

***Notice: This is the same motivation for Fenwick Trees, but Segment Trees will have some advantages / disadvantages*

WHY WOULD WE WANT THIS?

$$A = \{5, 10, 1, 11, 29, 3, 2, 209, 85, 6, 9, 11\}$$

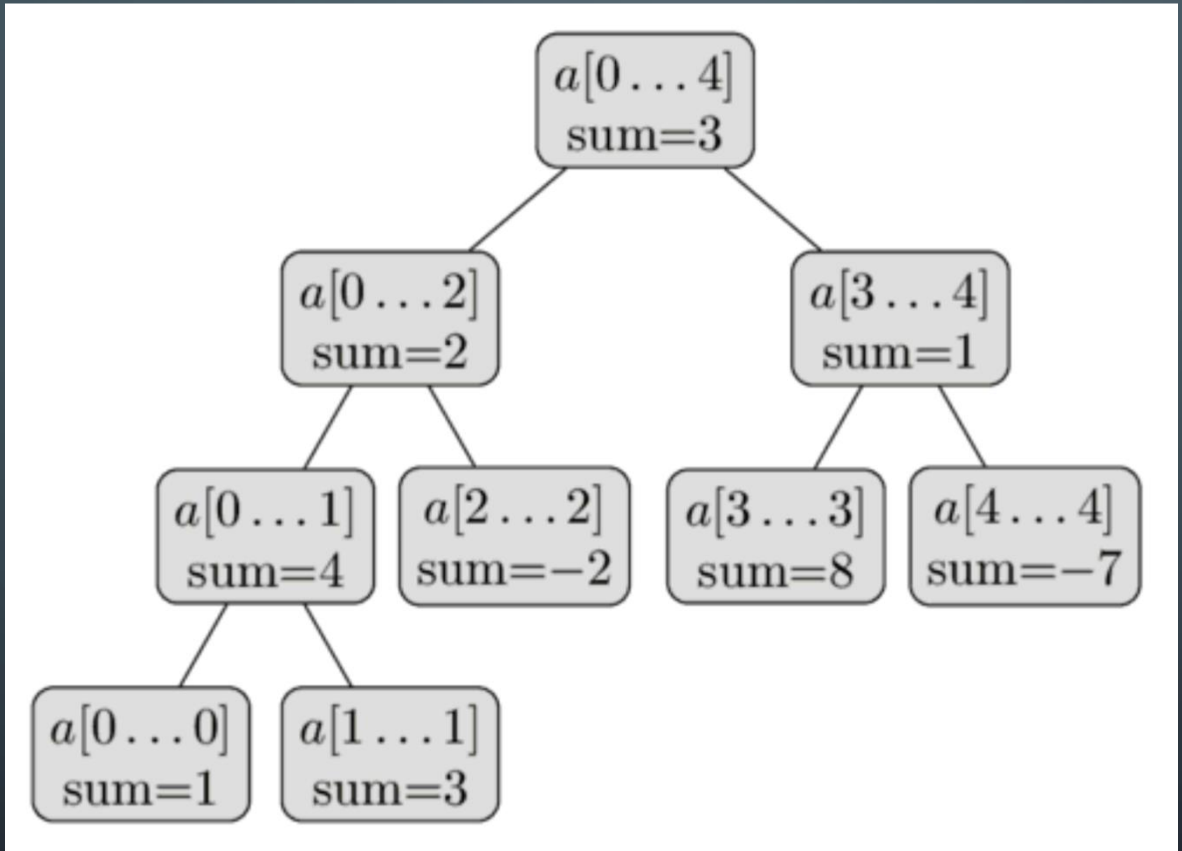
Suppose: That our stream of integers are stock prices over time, or sensor data across many time points, or sales per day (etc.).

Perhaps our company needs to be able to pull f() over any arbitrary range in A thousands of time per day

Suppose: That f() is a function we care about over any range. Max and min (for stock prices), average (for sensor data), or sum (for sales)

SEGMENT TREES

SEGMENT TREE



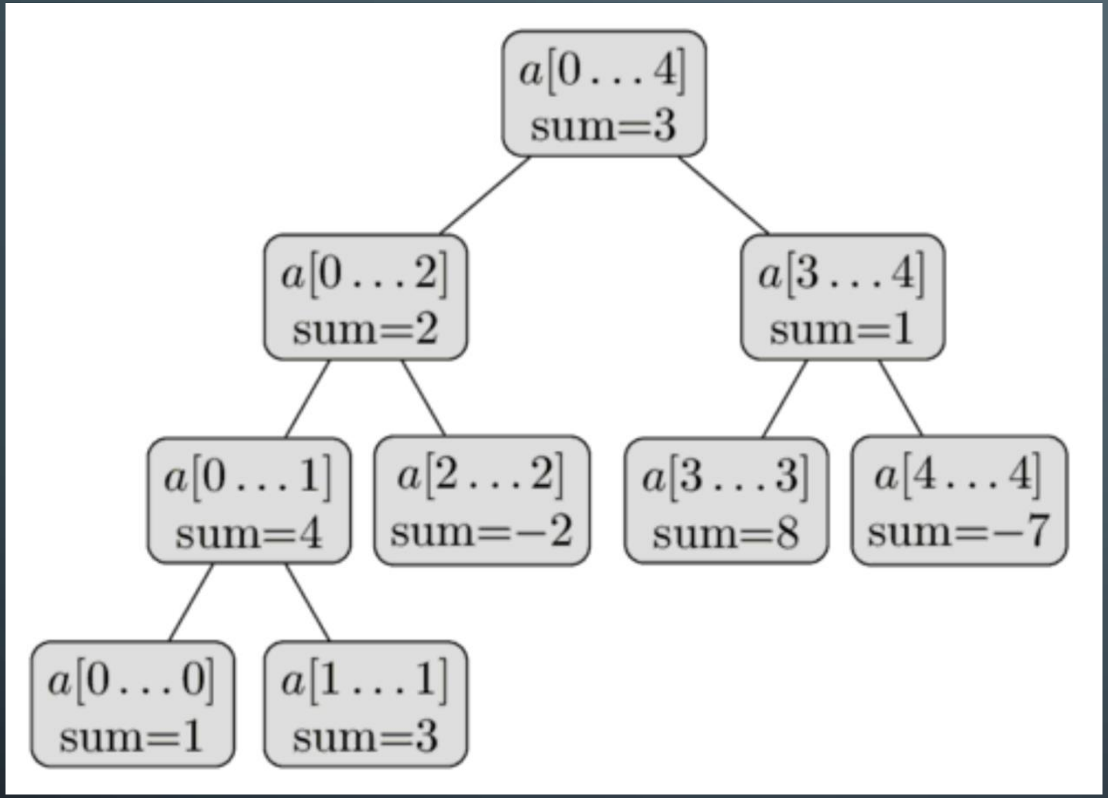
Here, we are representing the following array:

$A = \{1, 3, -2, 8, -7\}$

Each node stores the f() of interest value for a subrange of the array

Leaf nodes store f(), here we are using sum, for a single element. Usually trivial to compute.

SEGMENT TREE



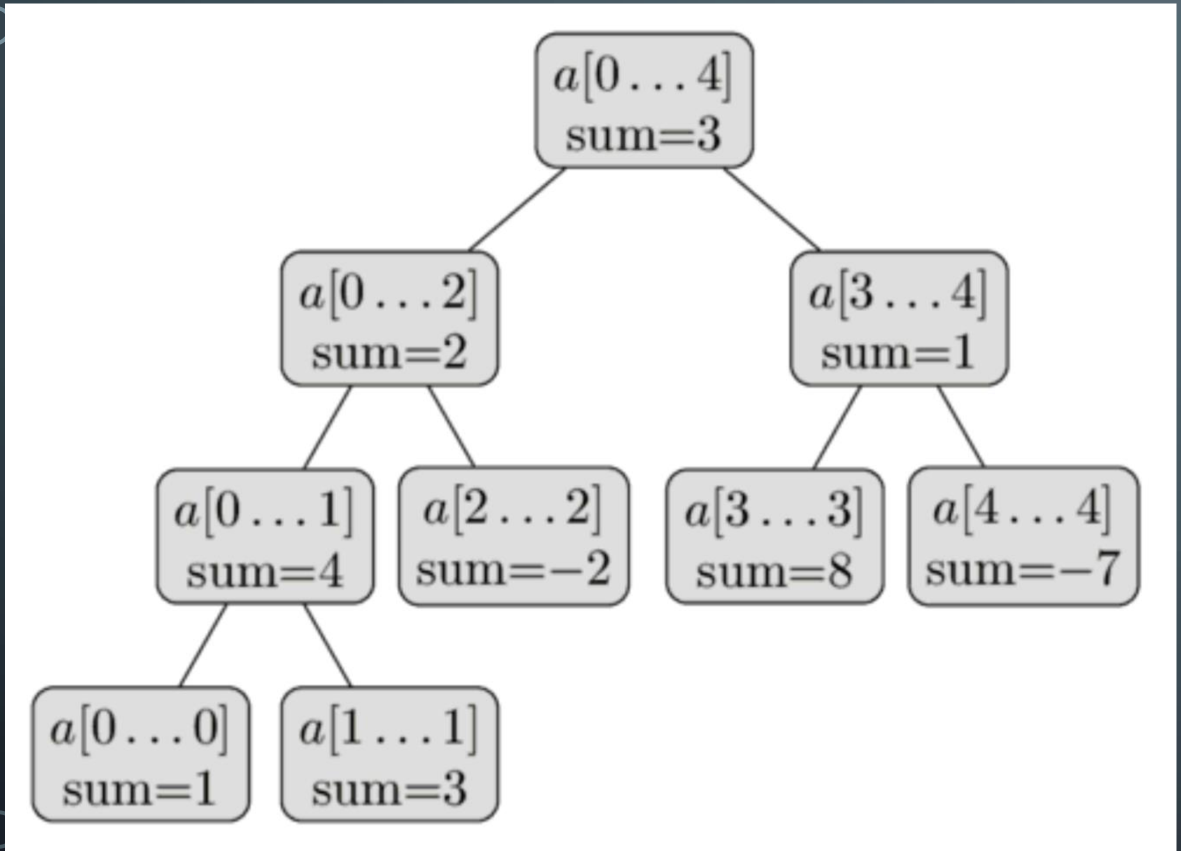
Here, we are representing the following array:

$A = \{1, 3, -2, 8, -7\}$

How much storage does this take (note that we don't store the sub-arrays, just the start and end indices, and f()

$$1 + 2 + 4 + 8 + \ \dots + 2^{(\log_2 n)+1} < 4n = \Theta(n)$$

CONSTRUCTING A SEGMENT TREE



Before constructing ST, need to determine two things:

1. The value(s) that gets stored in the nodes (e.g., the sum of the nodes in the given range)

2. The merge operation, which determines the value(s) from #1 above, given the value(s) of the two children.

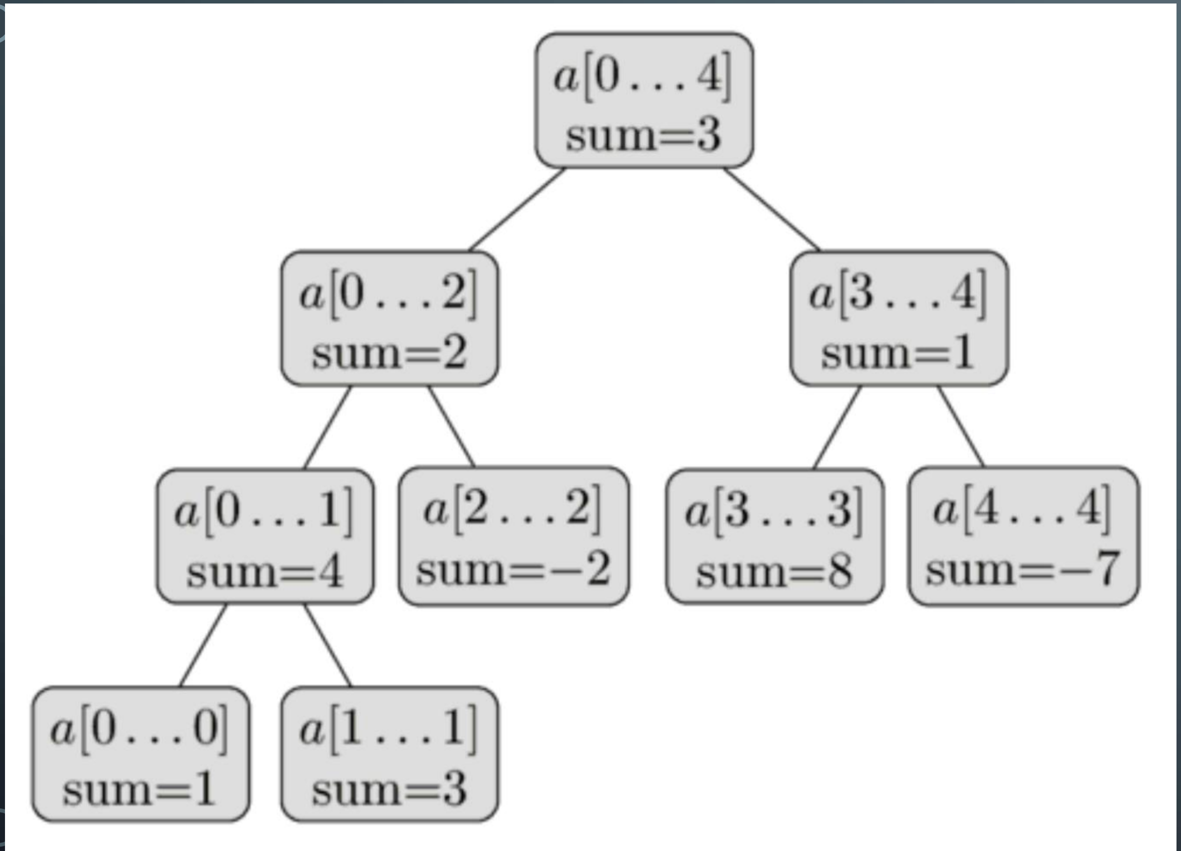
For sum, merge would simply be (child1 + child2)

CONSTRUCTING A SEGMENT TREE

Let's step through constructing this tree:

A = {1, 3, -2, 8, -7}

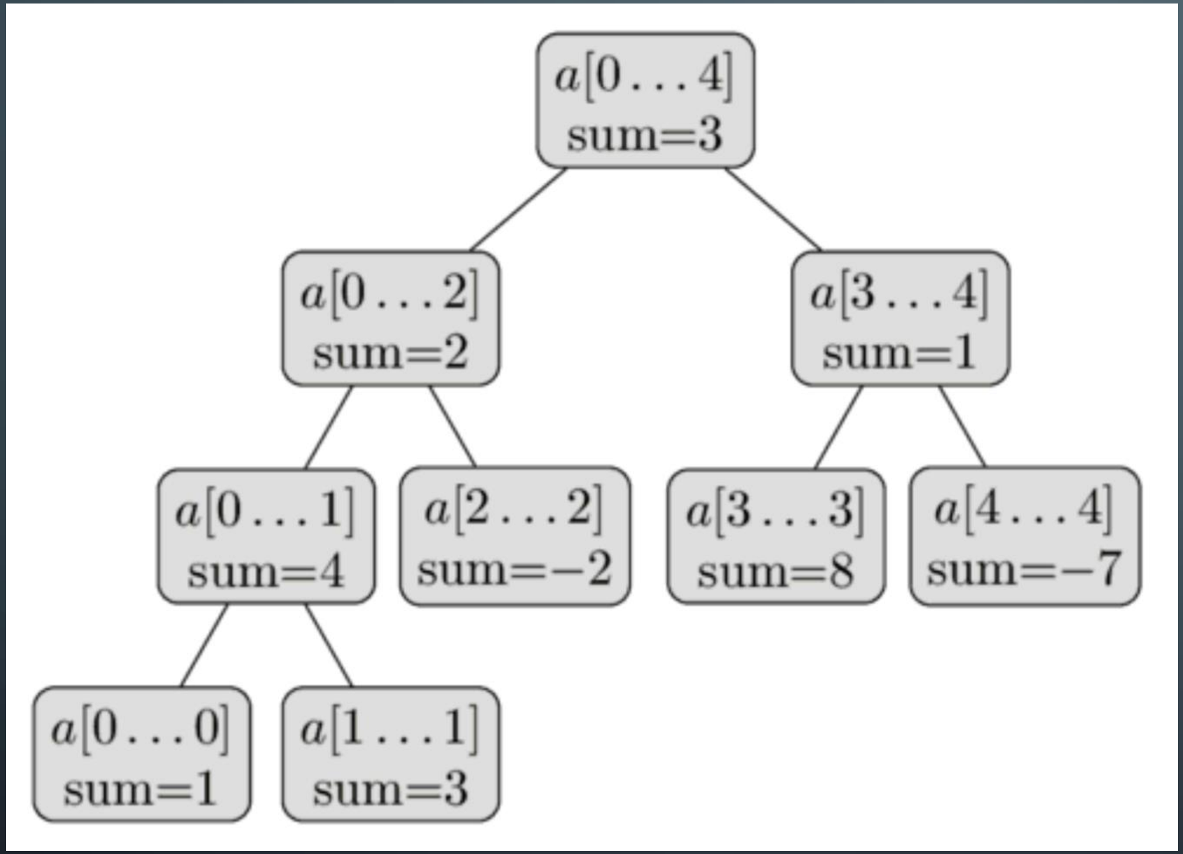
CONSTRUCTING A SEGMENT TREE



Psuedo-code for Segment Tree Construction:

```
Node:

   int left, right;
   int val;                    //can be different


ConstructSegTree(a[]):

   ConstructRecurse(a, new Node(0, a.size-1));


ConstructRecurse(a[], curNode):

   if(curNode.left == curNode.right):

      set curNode.val to base case value


   leftChild = new Node(curNode.left, mid);
   rightChild = new Node(mid+1, curNode.right);
   ConstructRecurse(a, leftChild);
   ConstructRecurse(a, rightChild);
   this.val = merge(leftChild, rightChild);
```

A QUICK NOTE ON REPRESENTATION



$T = \{3, 2, 1, 4, -2, 8, -7, 1, 3\}$

You can store Segment Trees the same way binary heaps are typically done. Use an array.

*Left child = ((index+1)*2) - 1*

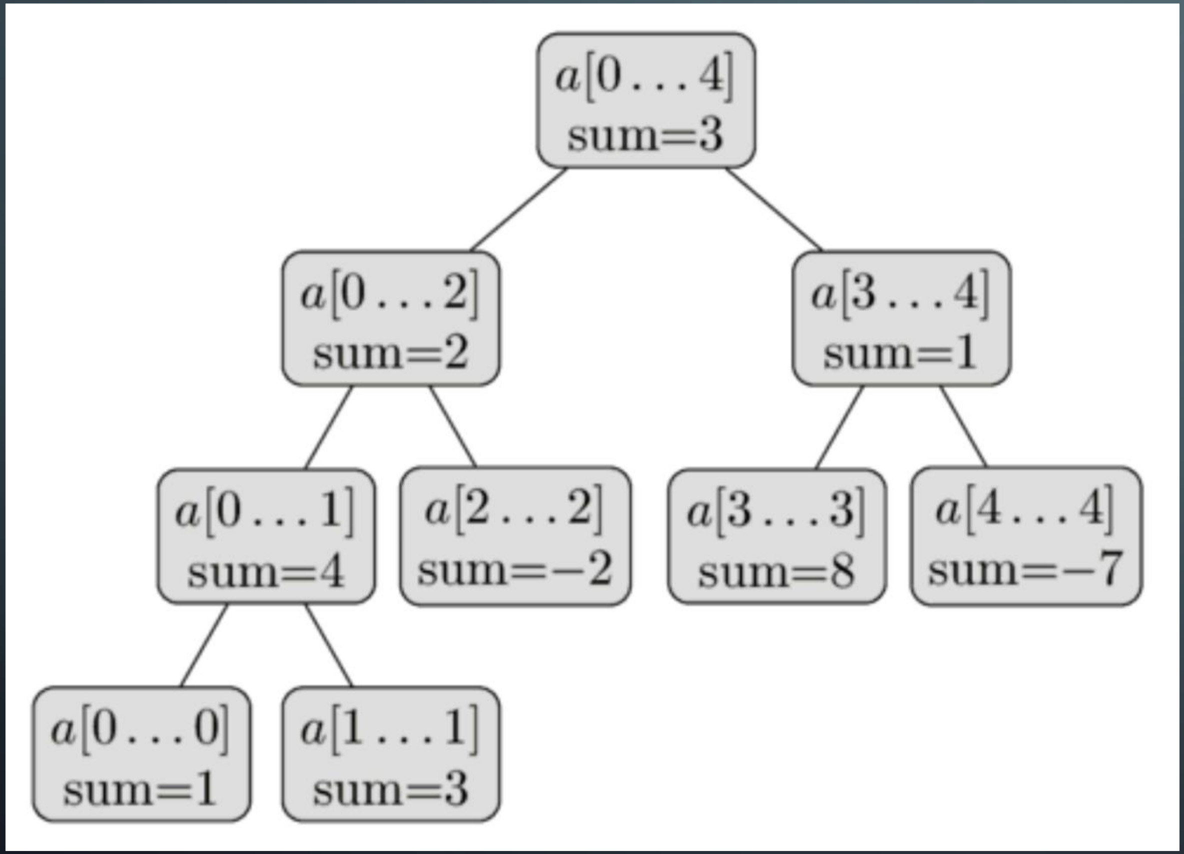
*Right child = (index+1)*2*

^^These are a tad simpler if you index by 1 instead of 0

I'll show these as trees in this slide deck though…

RANGE QUERIES

RANGE QUERIES



A = {1, 3, -2, 8, -7}

For now, let's assume that our function f() that we care about is sum (as in the example to the left)

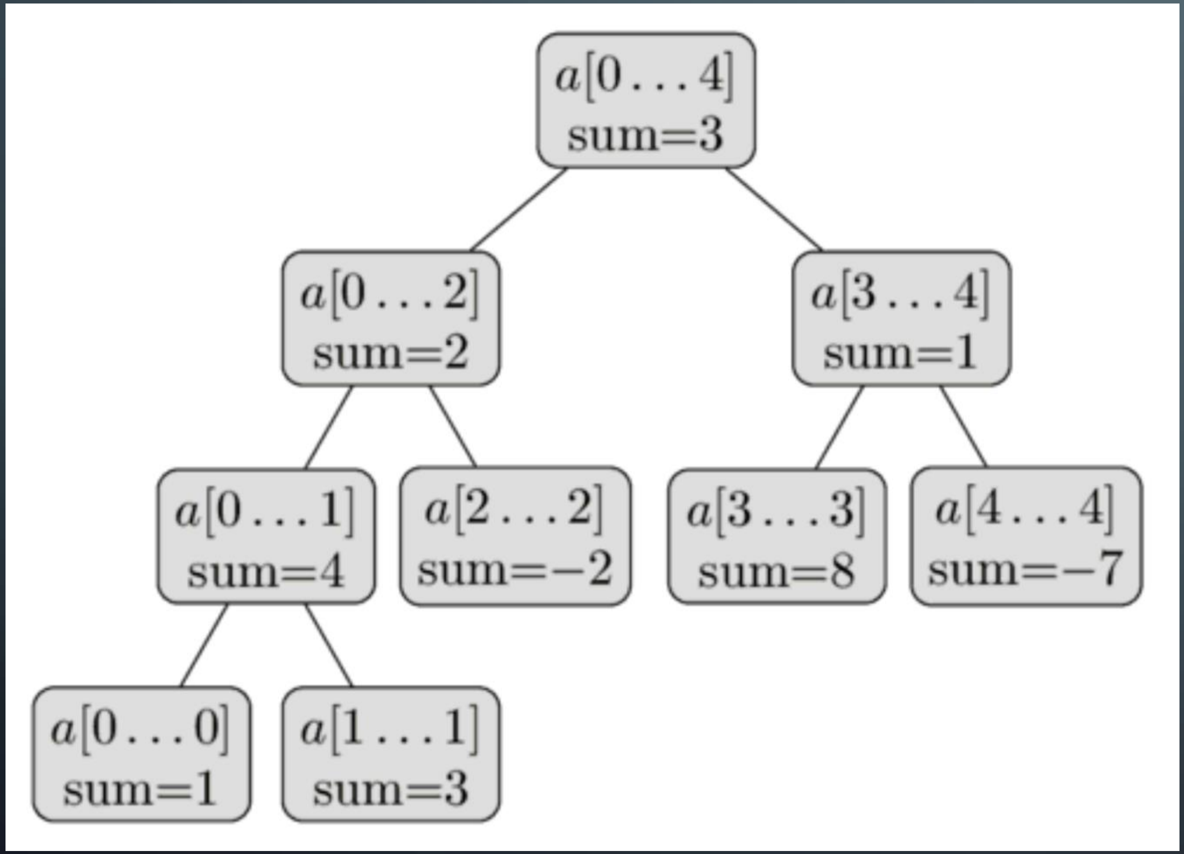
<u>We want to answer queries of the form</u>:

sum(left, right)

<u>Example</u>:

sum(2,4) = a[2]+a[3]+a[4] = -1

RANGE QUERIES



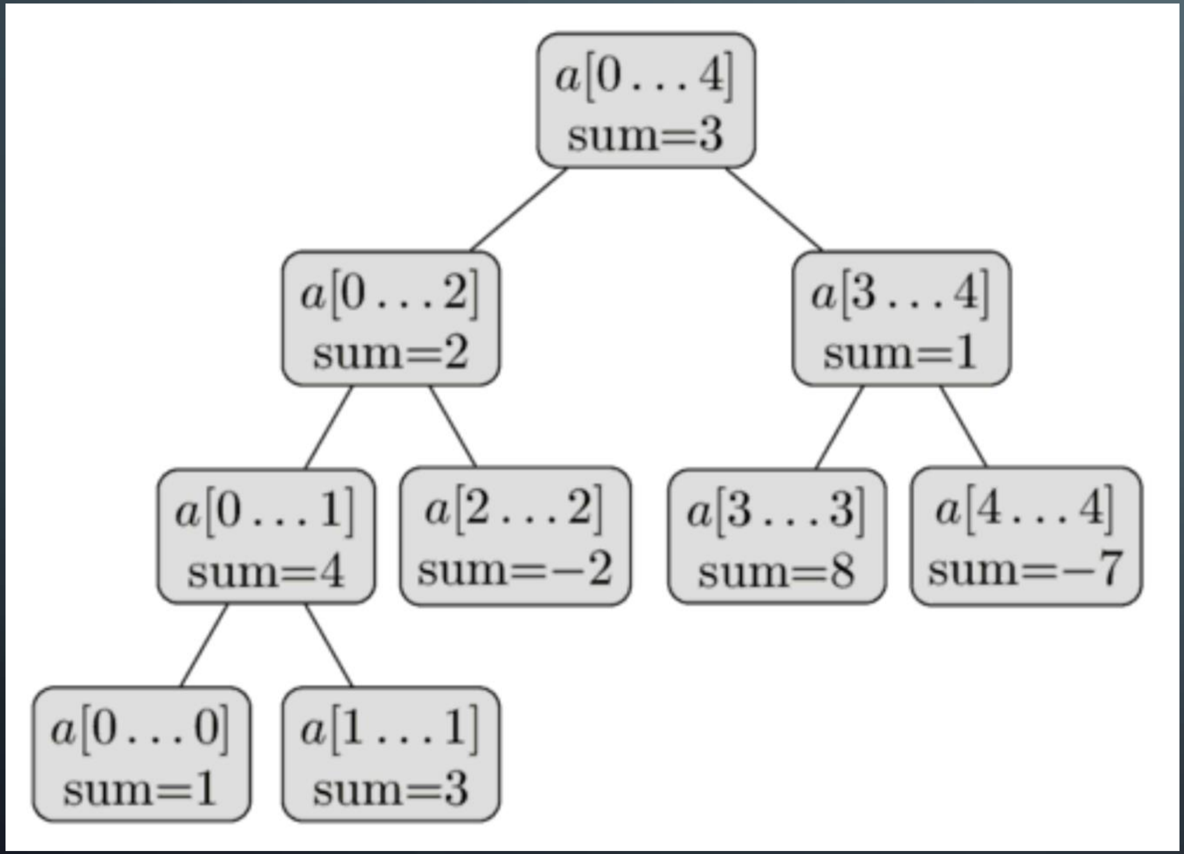
$A = \{1, 3, -2, 8, -7\}$

sum(left, right):

Three cases that could occur:

1. *[left, right] is exact range of this node*
2. *[left, right] falls completely within left child or right child*
3. *[left, right] crosses the dividing line of this node*

RANGE QUERY EXAMPLE

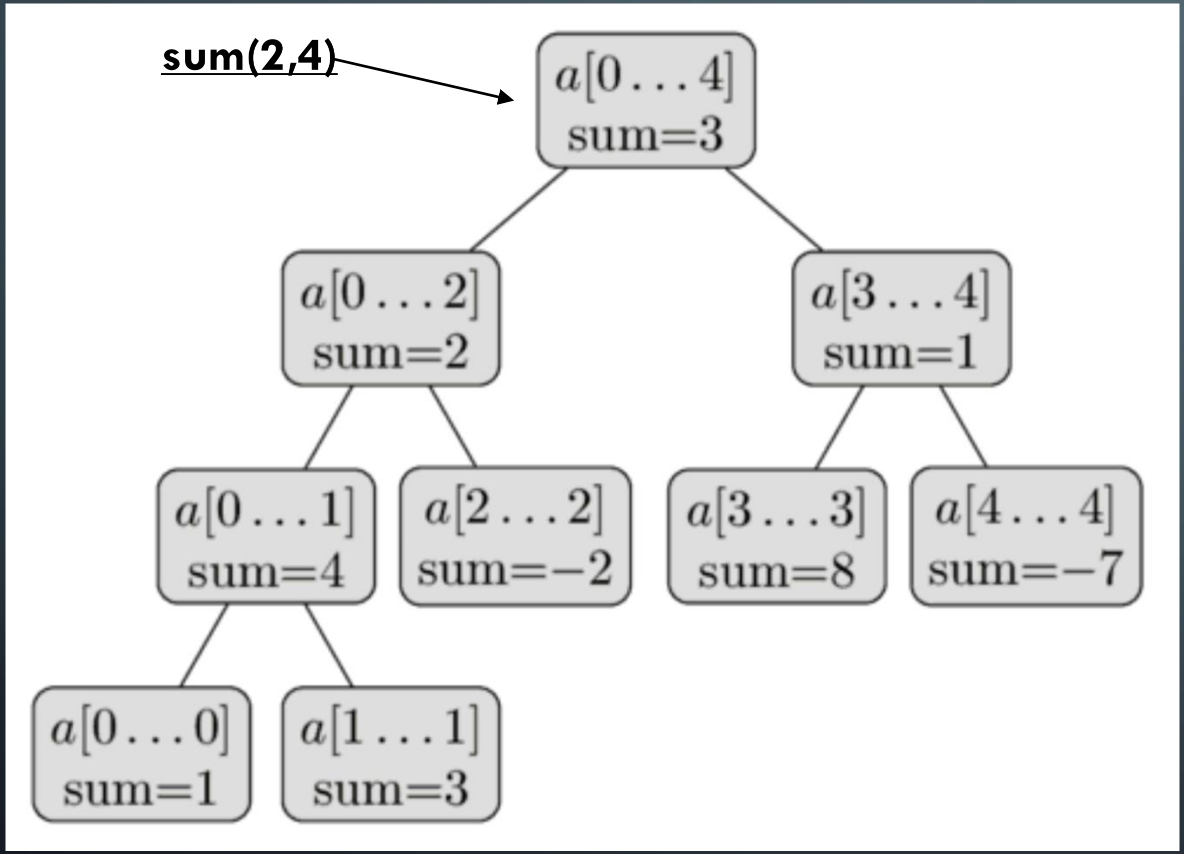


A = {1, 3, -2, 8, -7}

Let's step through:

sum(2, 4) *//answer should be -1*

RANGE QUERY EXAMPLE



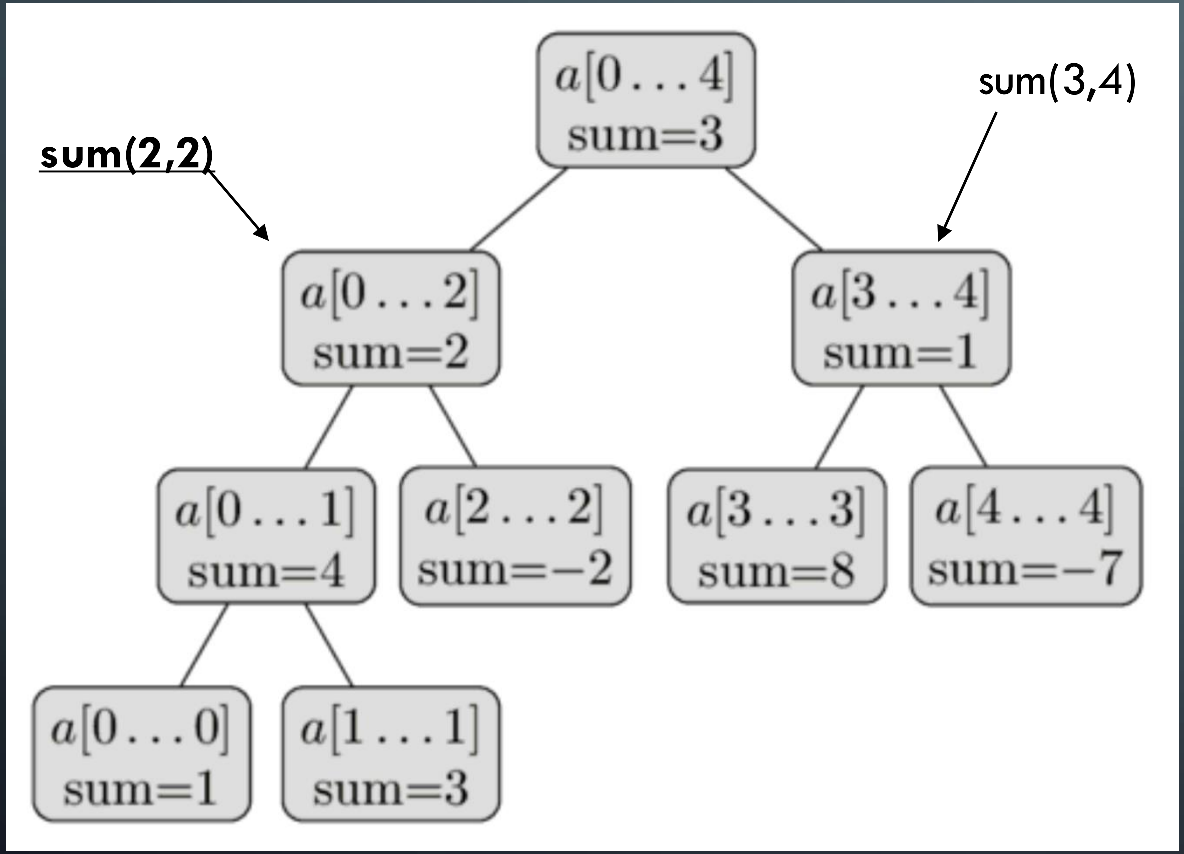
A = {1, 3, -2, 8, -7}

Let's step through:

sum(2, 4)//answer should be -1

[2,4] Spans both children, so recurse on both children!

RANGE QUERY EXAMPLE



A = {1, 3, -2, 8, -7}

Let's step through:

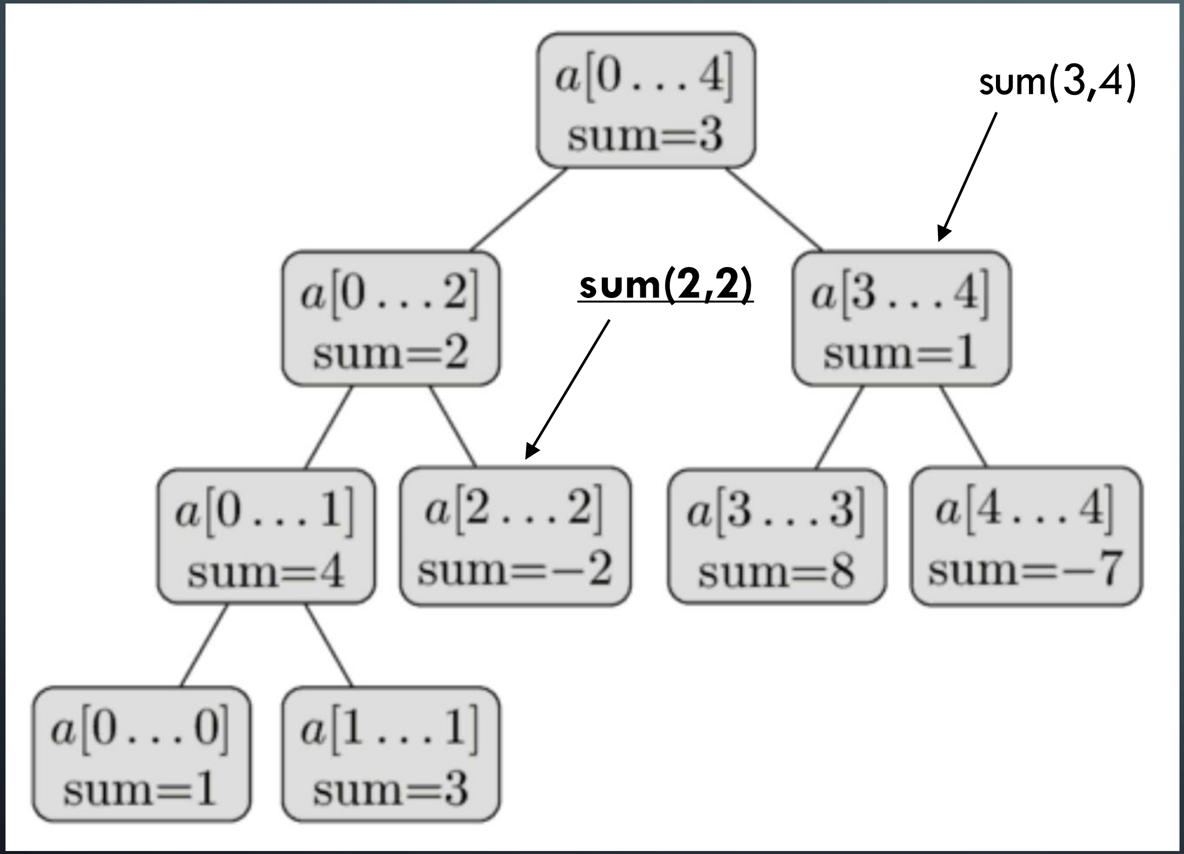
sum(2, 4) //answer should be -1

[2,4] Spans both children, so recurse on both children!

sum(2, 2)

[2,2] falls completely on right half, so recurse right!

RANGE QUERY EXAMPLE



A = {1, 3, -2, 8, -7}

Let's step through:

sum(2, 4) //answer should be -1

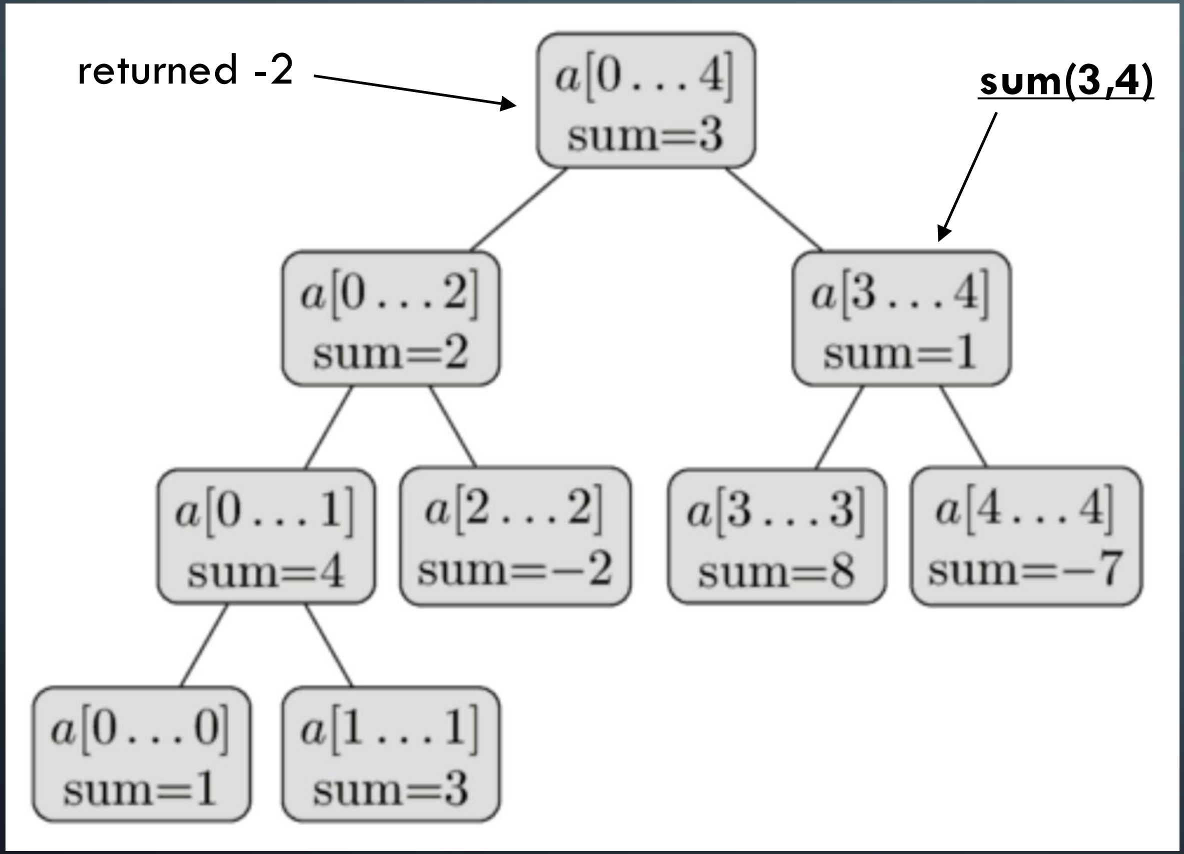
[2,4] Spans both children, so recurse on both children!

sum(2, 2)

[2,2] falls completely on right half, so recurse right!

[2,2] is now complete range, return -2

RANGE QUERY EXAMPLE



A = {1, 3, -2, 8, -7}

Let's step through:

sum(2, 4) //answer should be -1

[2,4] Spans both children, so recurse on both children!

sum(2, 2)

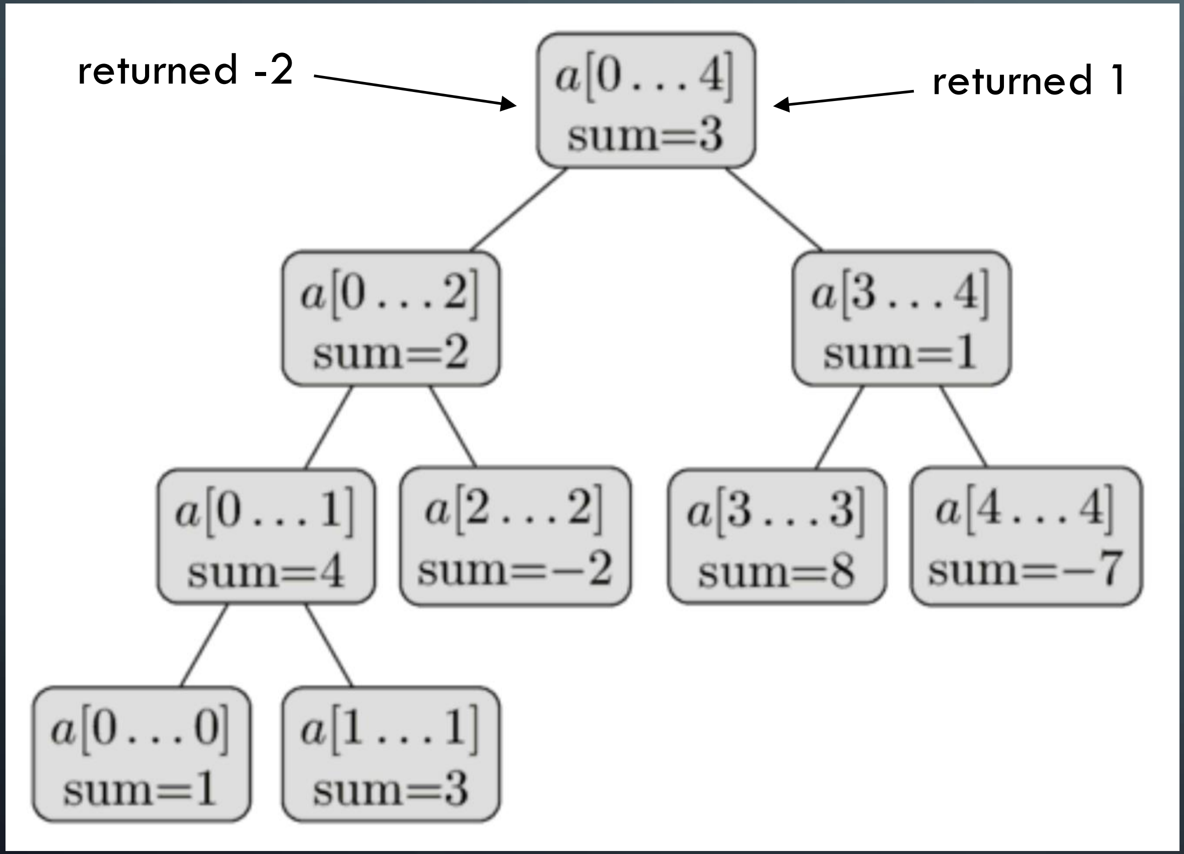
[2,2] falls completely on right half, so recurse right!

[2,2] is now complete range, return -2

sum(3, 4)

[3,4] is the entire range, so return 1

RANGE QUERY EXAMPLE



A = {1, 3, -2, 8, -7}

Let's step through:

sum(2, 4) *//answer should be -1*

[2,4] Spans both children, so recurse on both children!

sum(2, 2)

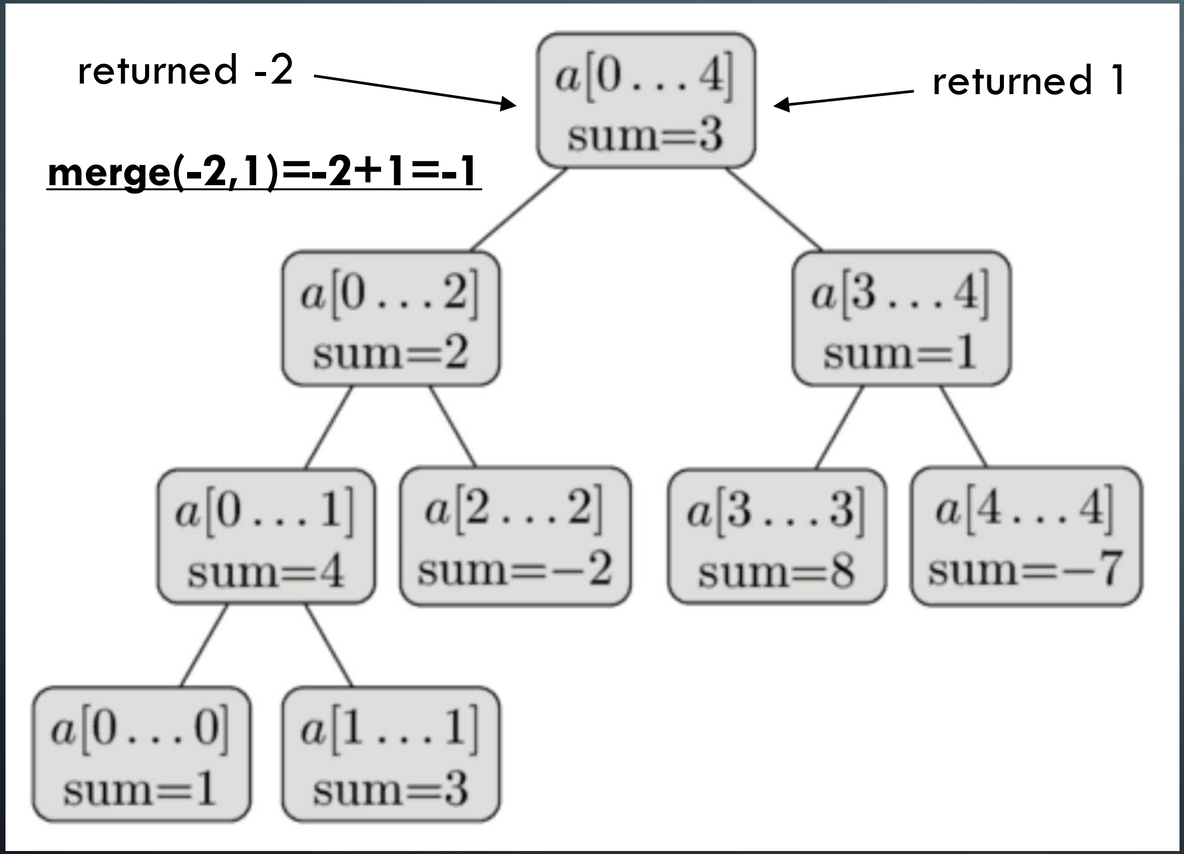
[2,2] falls completely on right half, so recurse right!

[2,2] is now complete range, return -2

sum(3, 4)

[3,4] is the entire range, so return 1

RANGE QUERY EXAMPLE



A = {1, 3, -2, 8, -7}

Let's step through:

sum(2, 4) *//answer should be -1*

[2,4] Spans both children, so recurse on both children!

sum(2, 2)

[2,2] falls completely on right half, so recurse right!

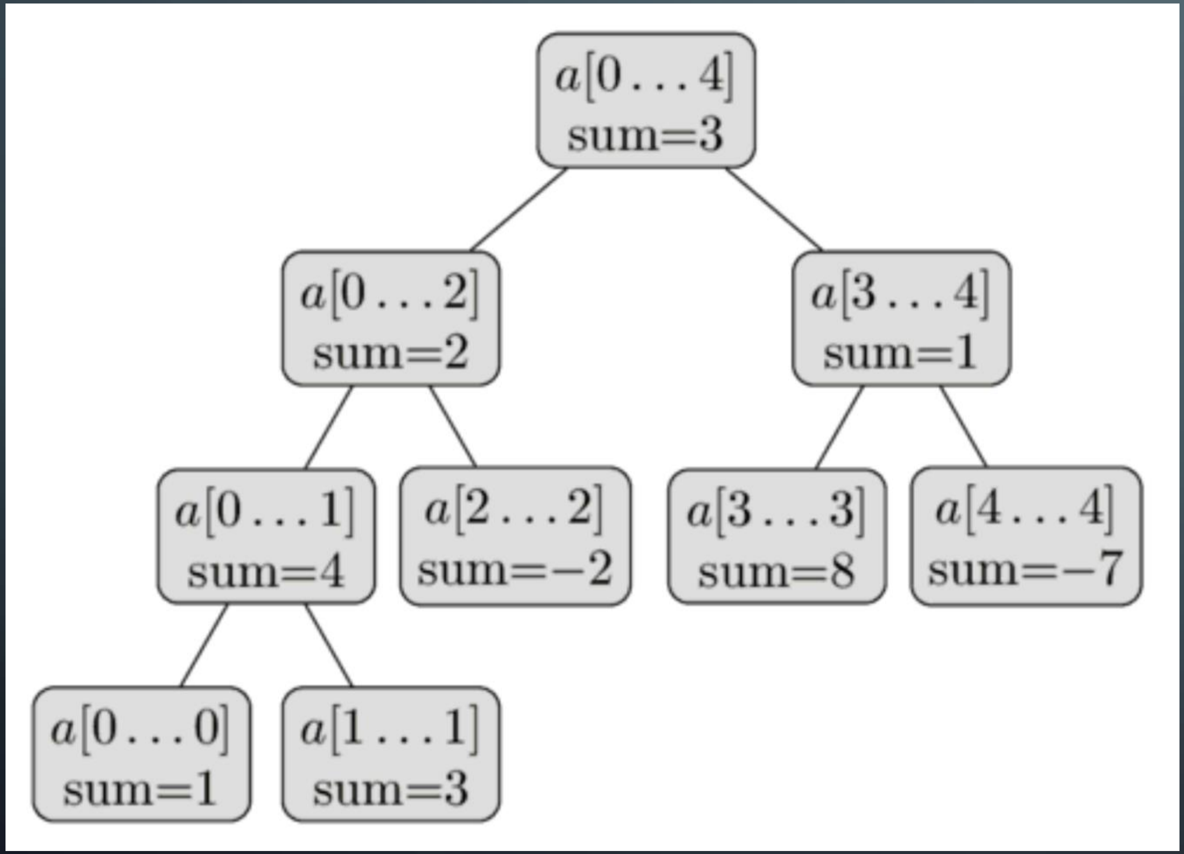
[2,2] is now complete range, return -2

sum(3, 4)

[3,4] is the entire range, so return 1

Finally, merge the two return values

THREE MORE EXAMPLES



A = {1, 3, -2, 8, -7}

Let's step through:

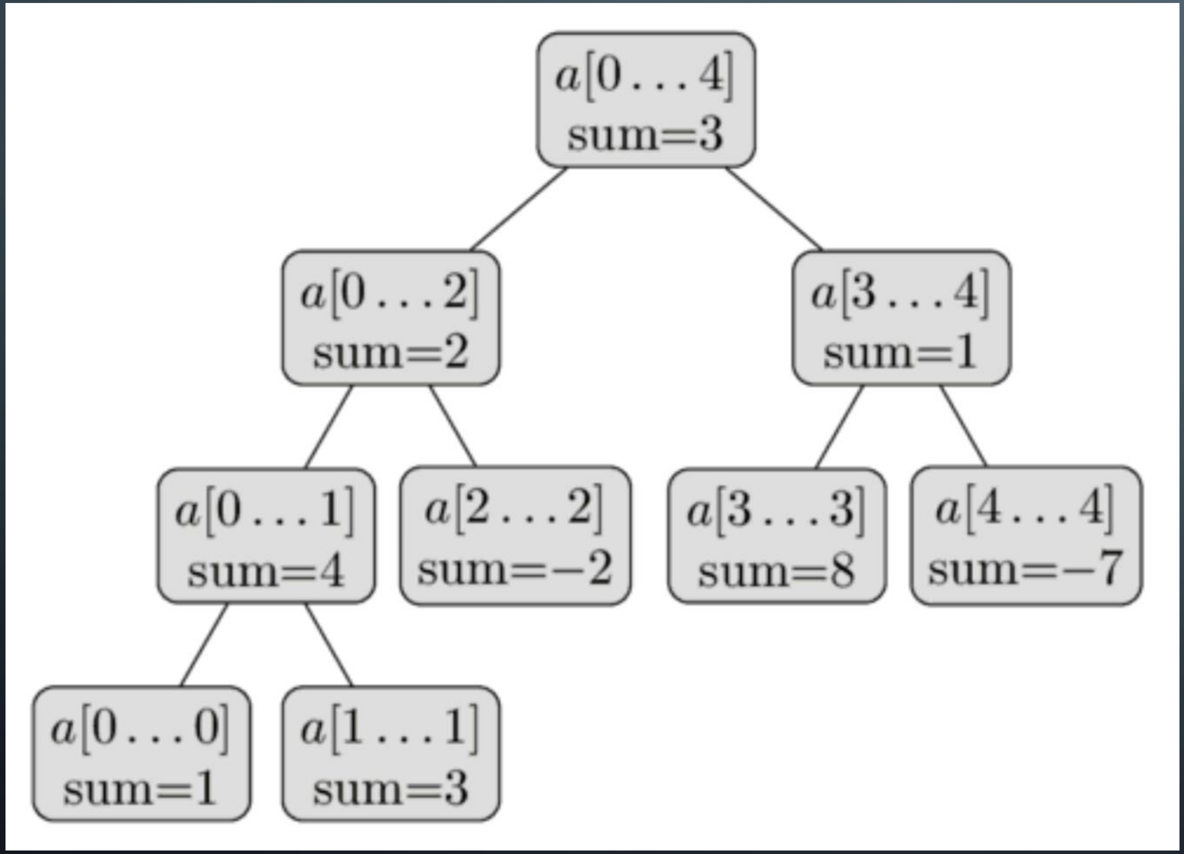
sum(0,4)

sum(0,1)

sum(1,3)

LET'S PSEUDOCODE THIS…

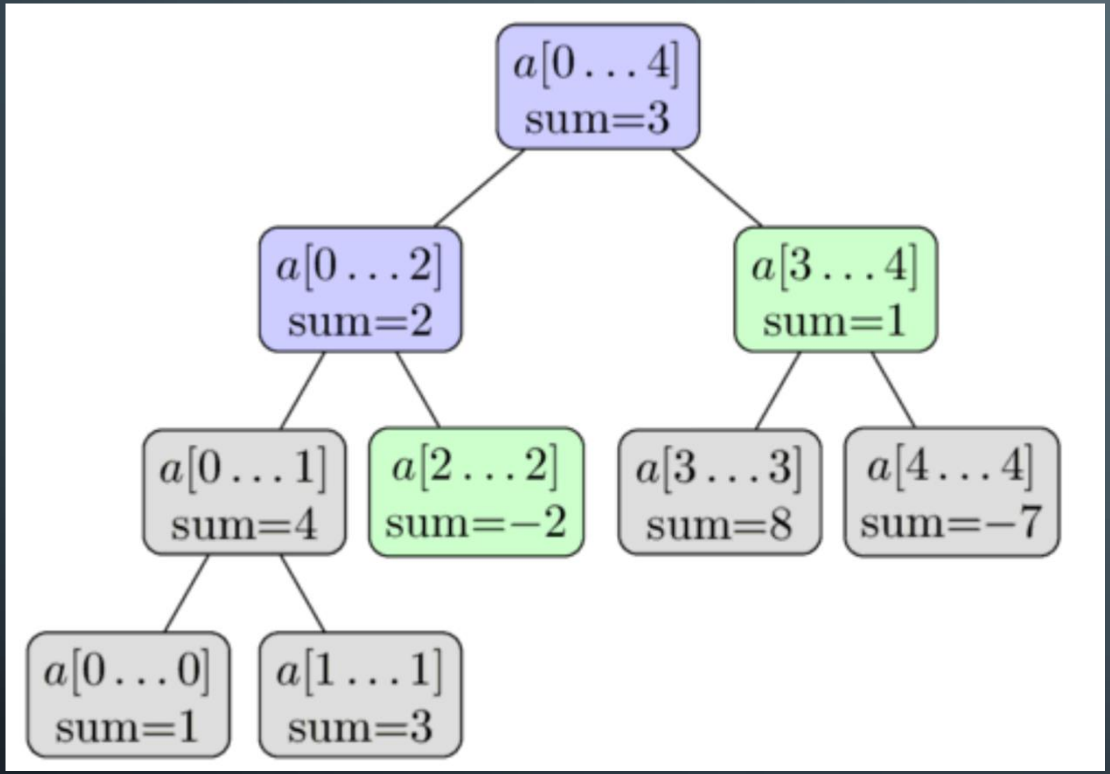
$A = \{1, 3, -2, 8, -7\}$




```
sumQuery(left, right):
    sumQuery(root, l, r)


sumQuery(curNode, l, r):
```

RANGE QUERY RUNTIME

RANGE QUERY RUNTIME



Another representation of the execution of sum(2, 4) from earlier.

$A = \{1, 3, -2, 8, -7\}$

What is the runtime of range queries?

Clearly it is $O(n)$ *and* $\Omega(\log n)$

Will this actually happen? Consider sum(0, 4)

Or will this happen? Always? Just sometimes? If latter, how often?

RANGE QUERY RUNTIME

<u>Claim</u>*: As a range query on a segment tree executes. The number of nodes explored (let's call it* e_i*) at each level (let's call it i) of the tree is at most 4.* $\forall_{i \geq 0} 0 \leq e_i \leq 4$

<u>Induction on the level of the tree as the algorithm executes.</u>

<u>Base Case</u>: Level 0 (root node). There is only one node, so $e_0 = 1 \leq 4$

RANGE QUERY RUNTIME

Claim: *As a range query on a segment tree executes. The number of nodes explored (let's call it* e_i*) at each level (let's call it i) of the tree is at most 4.* $\forall_{i \geq 0} 0 \leq e_i \leq 4$

Induction on the level of the tree as the algorithm executes.

Base Case: Level 0 (root node). There is only one node, so $e_0 = 1 \leq 4$

Inductive Hypothesis: Assume for some arbitrary level k, $e_k \leq 4$

RANGE QUERY RUNTIME

Claim: *As a range query on a segment tree executes. The number of nodes explored (let's call it e_i) at each level (let's call it i) of the tree is at most 4.* $\forall_{i \geq 0} 0 \leq e_i \leq 4$

Induction on the level of the tree as the algorithm executes.

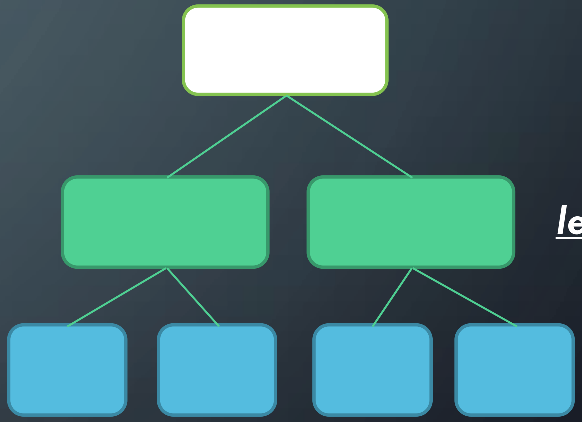
Base Case: Level 0 (root node). There is only one node, so $e_0 = 1 \leq 4$

Inductive Hypothesis: Assume for some arbitrary level k, $e_k \leq 4$

Inductive Step: Consider level k+1, There are two cases:

Case 1: Previous level has 2 or fewer recursive calls ($e_k \leq 2$)

Case 2: Previous level has 3 or 4 calls ($3 \leq e_k \leq 4$)



level k: $e_k = 2$

Because level k only has at most 2 nodes traversed, each can make at most 2 recursive calls, so $e_k \leq 4$

RANGE QUERY RUNTIME

Claim: *As a range query on a segment tree executes. The number of nodes explored (let's call it e_i) at each level (let's call it i) of the tree is at most 4.* $\forall_{i \geq 0} 0 \leq e_i \leq 4$

Induction on the level of the tree as the algorithm executes.

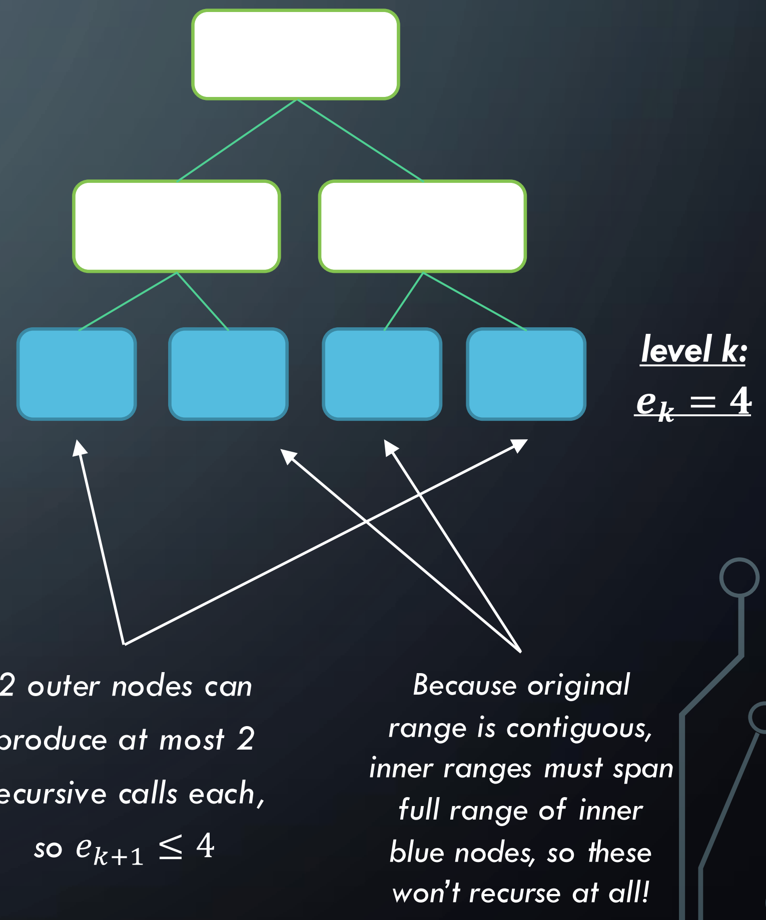
Base Case: Level 0 (root node). There is only one node, so $e_0 = 1 \leq 4$

Inductive Hypothesis: Assume for some arbitrary level k, $e_k \leq 4$

Inductive Step: Consider level k+1, There are two cases:

Case 1: Previous level has 2 or fewer recursive calls ($e_k \leq 2$)

Case 2: Previous level has 3 or 4 calls ($3 \leq e_k \leq 4$)



RANGE QUERY RUNTIME

***Claim**: As a range query on a segment tree executes. The number of nodes explored (let's call it e_i) at each level (let's call it i) of the tree is at most 4.* $\forall_{i \geq 0} 0 \leq e_i \leq 4$

Induction on the level of the tree as the algorithm executes.

Base Case: Level 0 (root node). There is only one node, so $e_0 = 1 \leq 4$

Inductive Hypothesis: Assume for some arbitrary level k, $e_k \leq 4$

Inductive Step: Consider level k+1, There are two cases:

Case 1: Previous level has 2 or fewer recursive calls ($e_k \leq 2$)

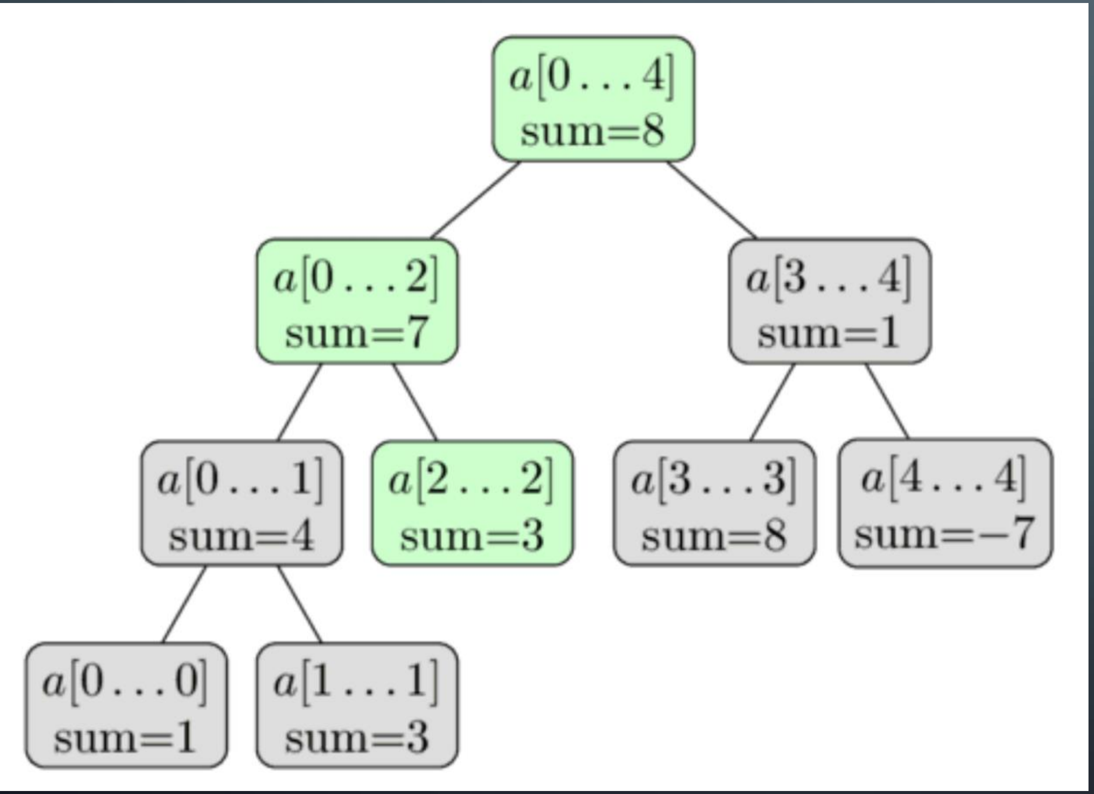
Case 2: Previous level has 3 or 4 calls ($3 \leq e_k \leq 4$)

Claim is proven! Because each level has at most 4 nodes traversed, and there are log(n) levels of the segment tree, the total nodes traversed is bounded by $\sum_{e_i} 4 \log n$

UPDATE QUERIES

UPDATE QUERY

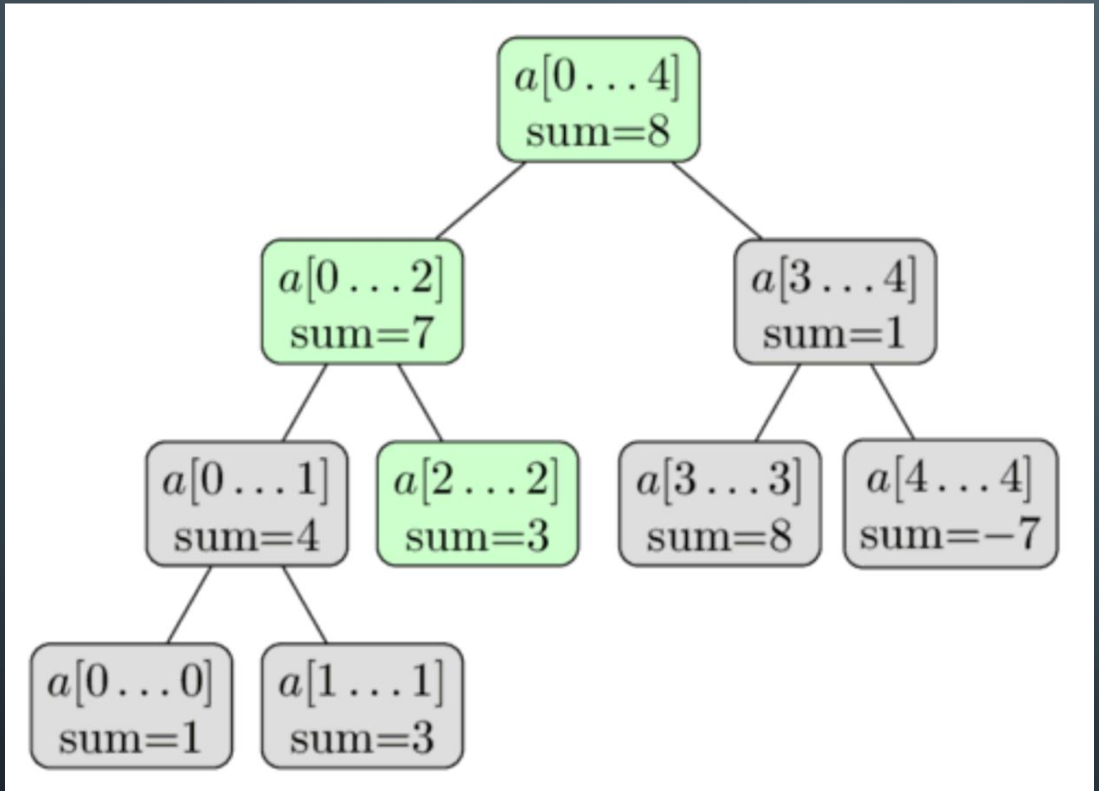
$A = \{1, 3, -2, 8, -7\}$ $\xrightarrow{update(2,\ 3)}$ $A = \{1, 3, 3, 8, -7\}$



Traverse the tree to find index 2

Make sure to call merge() to update value along the path as recursion returns

TWO MORE QUICK EXAMPLES



$A = \{1, 3, 3, 8, -7\}$

Let's step through:

update(3, 10)

update(0, 2)

MORE EXAMPLES

NUMBER OF 0'S AND K'TH 0

Given an array, we want to be able to query for the number of zero's in any segment (range) AND also query for the location of the k'th zero.

$A = \{1, 0, 0, -2, 0, 7, 0, 0, 3, 3, 1, 0, 1, 0, 0, 0, 0, 4\}$

1. *What will we store in each node of the segment tree?*
2. *How will we define the merge() operation?*
3. *How will we query for the k'th zero?*

Try to solve this problem on your own!

NUMBER OF 0'S AND K'TH 0

Given an array, we want to be able to query for the number of zero's in any segment (range) AND also query for the location of the k'th zero.

A = {1, 0, 0, -2, 0, 7, 0, 0, 3, 3, 1, 0, 1, 0, 0, 0, 0, 4}

Store: The number of 0's in each segment in the node itself

Merge: Simple, just add number of 0's together

Query (# of 0's): Simply query like we did with sum()

Query (k'th 0): If number of 0's on left is k or larger, recurse left. Or search for the leftChild-k on right

Update: Only thing that matters is…did the new value become a 0 or change FROM a 0. Update num 0's by 1 if so and merge up.

OTHER SEGMENT TREE EXAMPLES

FINDING MAXIMAL SUB-SEGMENT

Given an array A, and a range [l .. r], find the subsegment [l' .. r'] such that $l \leq l'$ and $r \geq r'$ and sum of [l' .. r'] is maximal

A = {1, 6, -3, -1, -1, 1, -5, 11, 12, 1, 1, 2, -4, -2, -8, 9, 11}

So maximal(2, 8) would return 23 because range [7 .. 8] (11+12) is the maximal sum within the range [2 .. 8].

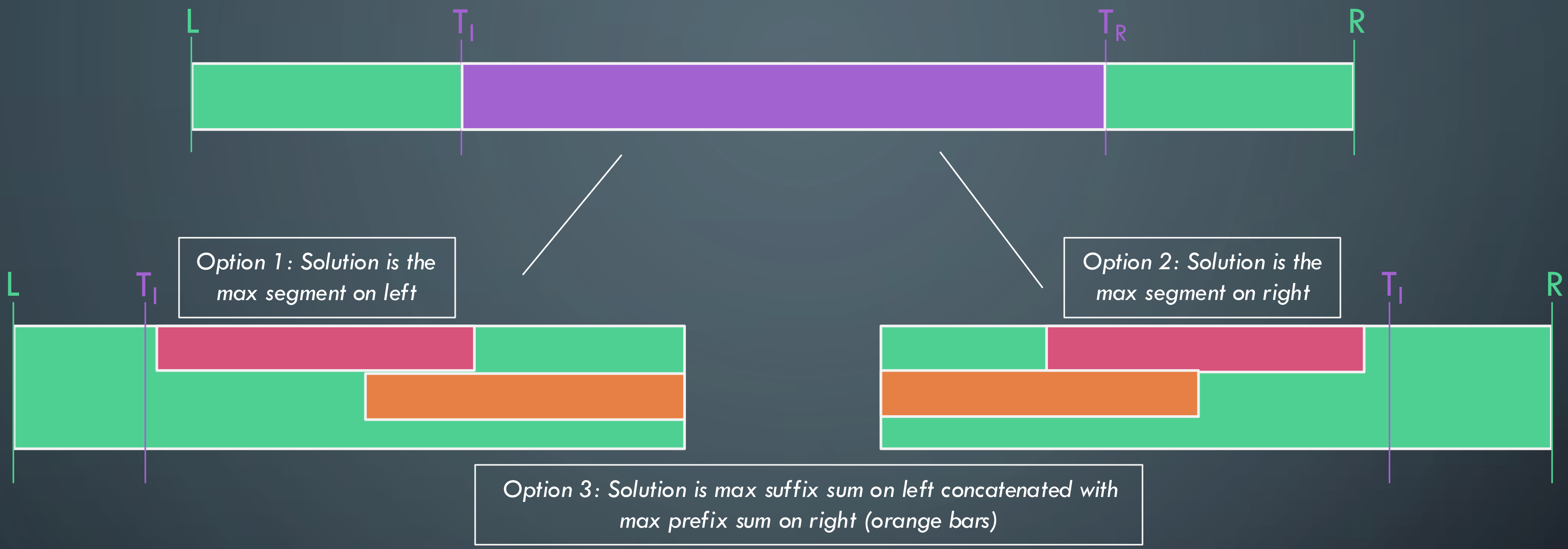
1. *What will we store in each node of the segment tree?*
2. *How will we define the merge() operation?*
3. *How will we query?*

FINDING MAXIMAL SUB-SEGMENT

Given an array A, and a range [l .. r], find the subsegment [l' .. r'] such that $l \leq l'$ and $r \geq r'$ and sum of [l' .. r'] is maximal

Key Idea: *Observe that the maximal sub-segment of a range is one of a few options given the solution to its two children.*

FINDING MAXIMAL SUB-SEGMENT



Each node should store:

<u>Sum of segment</u>: useful for computing other values below

<u>Max prefix sum</u>: for combining across splits

<u>Max suffix sum</u>: for combining across splits

<u>Ans</u>: The answer (max segment)

How to merge:

$sum = L.sum + R.sum$

$pre = Max(L.pre,\ L.sum + R.pre)$

$suf = Max(R.suf,\ R.sum + L.suf)$

$ans = Max(L.ans,\ R.ans,\ L.suf + R.pre)$

CONCLUSION

CONCLUSIONS

Strengths:

- More intuitive than Fenwick trees for many applications
- Binary Trees very easy to implement
- Seems to work for a wider array of functions of interest
- Only really have to focus on what to store and merge function, once you get those right the rest falls into place.

***Weaknesses*:**

- Uses a bit more memory than Fenwick Tree (because Fenwick tree isn't really a tree and doesn't have internal nodes
- Does not take advantage of fast bit operations to traverse tree