BuildTree (1, r, i):

- if (1 == r):

- tree[i] = arr[1]

- return

- mid = (1+r)/2

- BuildTree (1, m, i\*2+1)

- BuildTree (m+1, r, i\*2+2)

- tree[i] = [tree[i\*2+1] + tree[i\*2+2]

Update (ind, new\_val, 1, r, i):

- if (1 == r):

- tree[i] = new\_val

- return

- m = (1+r)/2

- if (m >= ind) Update (ind, new\_val, 1, m, i\*2+1)

- else Update (ind, new\_val, m+1, r, i\*2+2)

- tree[i] = [tree[i\*2+1] + tree[i\*2+1]

Query (x, y, 1, r, i):

- if (r < x || 1 > y) return 0

- if (1 >= x && r <= y)

- return tree[i]

- m = (1+r)/2

- return Query(1, m, i\*2+1) + Query (m+1, r, i\*2+2)

**Iterative Segment Tree (Simplified Approach)**

**Note:** This approach uses a 0-indexed array for the original data (arr[0...n-1]) but builds the tree in a 1-indexed array (tree[1...2\*n]) for simpler indexing. The leaves of the tree are stored in tree[n] to tree[2\*n - 1].

**BuildTree\_Iterative(n):**

* base = n *// The first leaf node is at index n*
* **For** i from 0 to n-1:
  + tree[base + i] = arr[i] *// Copy all elements to the leaf level*
* **For** i from n-1 down to 1:
  + tree[i] = tree[2\*i] + tree[2\*i + 1] *// Build the tree from the bottom up*

**Update\_Iterative(ind, new\_val):**

* pos = n + ind *// Find the leaf node corresponding to index ind*
* tree[pos] = new\_val *// Update the leaf's value*
* **While** pos > 1:
  + pos = pos / 2 *// Move to the parent node*
  + tree[pos] = tree[2\*pos] + tree[2\*pos + 1] *// Recalculate the parent's value from its children*

**Query\_Iterative(x, y):**

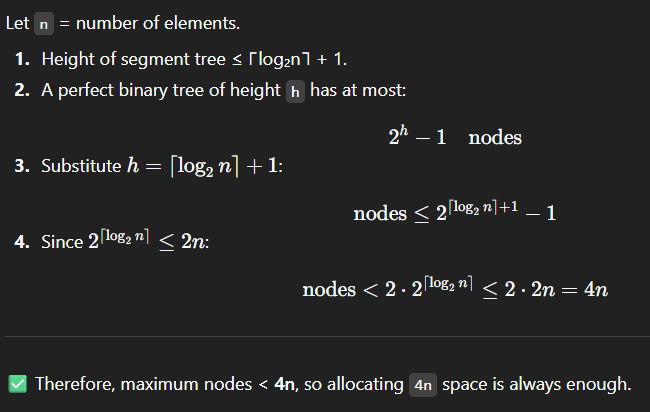
* left = n + x *// Get the leaf node for the left bound*
* right = n + y *// Get the leaf node for the right bound*
* sum = 0
* **While** left <= right:
  + **If** left is odd (*it's a right child*):
    - sum += tree[left] *// Add the value and move right*
    - left = left + 1
  + **If** right is even (*it's a left child*):
    - sum += tree[right] *// Add the value and move left*
    - right = right - 1
  + left = left / 2 *// Move both bounds up to the next level*
  + right = right / 2
* **Return** sum

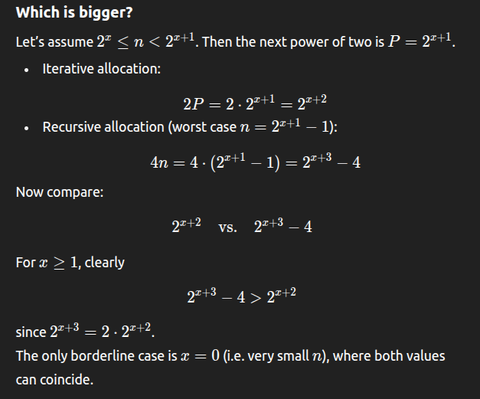
**Why 4n is enough for segment tree array size?**

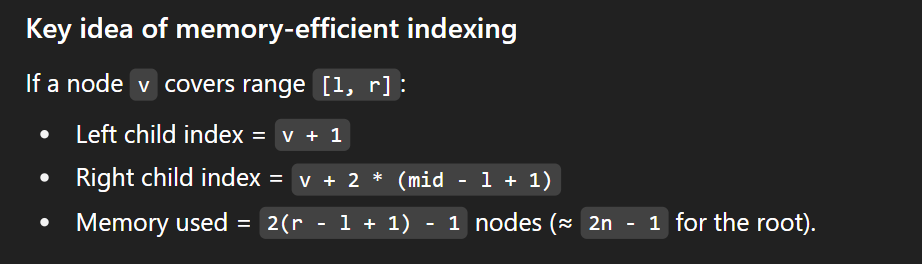
* A segment tree is a **binary tree** built on an array of size n.
* Worst case: if n is not a power of 2, the tree is padded to the next power of 2.
* A **perfect binary tree** with n leaves has fewer than 2 \* 2n = 4n total nodes.
* So 4n space always covers all possible cases.

**Example (n = 5):**

* Next power of 2 ≥ 5 is 8.
* A perfect binary tree with 8 leaves has 2\*8 - 1 = 15 nodes.
* 4n = 20, which is **more than 15**, so it’s enough.







Memory Efficient Approach apparently:

BuildTree(v, l, r):

if l == r:

tree[v] = arr[l]

return

mid = (l + r) // 2

left = v + 1

right = v + 2 \* (mid - l + 1)

BuildTree(left, l, mid)

BuildTree(right, mid+1, r)

tree[v] = tree[left] + tree[right]

Update(v, l, r, ind, new\_val):

if l == r:

tree[v] = new\_val

return

mid = (l + r) // 2

left = v + 1

right = v + 2 \* (mid - l + 1)

if ind <= mid:

Update(left, l, mid, ind, new\_val)

else:

Update(right, mid+1, r, ind, new\_val)

tree[v] = tree[left] + tree[right]

Query(v, l, r, ql, qr):

if r < ql or l > qr:

return 0 // no overlap

if ql <= l and r <= qr:

return tree[v] // total overlap

mid = (l + r) // 2

left = v + 1

right = v + 2 \* (mid - l + 1)

return Query(left, l, mid, ql, qr) + Query(right, mid+1, r, ql, qr)

Recursive with 0 indexed tree array

