```
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.

Let `n` = number of elements.

1. Height of segment tree ≤ $\lceil \log_2 n \rceil + 1$.
2. A perfect binary tree of height `h` has at most:
$$2^h - 1 \quad \text{nodes}$$
3. Substitute $h = \lceil \log_2 n \rceil + 1$:
$$\text{nodes} \leq 2^{\lceil \log_2 n \rceil + 1} - 1$$
4. Since $2^{\lceil \log_2 n \rceil} \leq 2n$:
$$\text{nodes} < 2 \cdot 2^{\lceil \log_2 n \rceil} \leq 2 \cdot 2n = 4n$$

✅ Therefore, maximum nodes < **4n**, so allocating `4n` space is always enough.

Which is bigger?

Let's assume $2^x \le n < 2^{x+1}$. Then the next power of two is $P = 2^{x+1}$.

- Iterative allocation:

$$2P = 2 \cdot 2^{x+1} = 2^{x+2}$$

- Recursive allocation (worst case $n = 2^{x+1} - 1$):

$$4n = 4 \cdot (2^{x+1} - 1) = 2^{x+3} - 4$$

Now compare:

$$2^{x+2} \quad \text{vs.} \quad 2^{x+3} - 4$$

For $x \ge 1$, clearly

$$2^{x+3} - 4 > 2^{x+2}$$

since $2^{x+3} = 2 \cdot 2^{x+2}$.
The only borderline case is $x = 0$ (i.e. very small n), where both values can coincide.

Key idea of memory-efficient indexing

If a node `v` covers range `[l, r]`:

- Left child index = `v + 1`
- Right child index = `v + 2 * (mid - l + 1)`
- Memory used = `2(r - l + 1) - 1` nodes (≈ `2n - 1` for the root).

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

```
BuildTree (l, r, i):

    - if (l == r):

        - tree[i] = arr[l]

        - return

    - mid = (l+r)/2

    - BuildTree (l, 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, l, r, i):

    - if (l == r):

        - tree[i] = new_val

        - return

    - m = (l+r)/2

    - if (m >= ind) Update (ind, new_val, l, 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, l, r, i):

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

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

        - return tree[i]

    - m = (l+r)/2

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