Exercise 2. Show that the optimal choice is $b = \Theta(\sqrt{n})$.

The time per operation is:

$$T(b) = (n/b) + b,$$

Where (n/b) is the number of blocks and b is the leftover size.

To minimize b, we take the derivative wrt b and set it to zero:

$d/db\,(n/b + b) = -n/b^2 + 1 = 0$
$n/b^2 = 1$
$b^2 = n$
$b = \sqrt{n}$

Thus, the optimal choice is $b = \Theta(\sqrt{n})$, giving $T(b) = O(\sqrt{n})$.

Exercise 3. Show that the optimal choice is $b = \Theta(n^{1/3})$ and $c = \Theta(n^{1/3})$.

The time per operation is $T(b, c) = (n/bc) + b + c$, where (n/bc) is the number of big blocks, b is the small block size, and c is the number of small blocks per big block.

To minimize, we set the partial derivatives to zero:

$$\frac{\partial T}{\partial b} = -\frac{n}{b^2c} + 1 = 0 \quad \Rightarrow \quad \frac{n}{b^2c} = 1 \quad \Rightarrow \quad b^2c = n$$

$$\frac{\partial T}{\partial c} = -\frac{n}{bc^2} + 1 = 0 \quad \Rightarrow \quad \frac{n}{bc^2} = 1 \quad \Rightarrow \quad bc^2 = n$$

Let $b^2c = bc^2 \Rightarrow b = c$.
substituting $b = c$ into $b^2c = n$:
$b^3 = n$
$b = n^{1/3}$
With the optimal choice giving $T(b,c) = O(n^{1/3})$.

Exercise 4. Show that a segment tree representing an array of length n has $2n - 1$ nodes. Hint: Note that each internal node has exactly two children.

A segment tree for an array of length n is a full binary tree, meaning each internal node has exactly two children, with n leaves.

Let L = n be the number of leaves. In a full binary tree, number of internal nodes I = L - 1.

so the total nodes:

$$N = L + I = n + (n - 1) = 2n - 1.$$

Thus, a segment tree has exactly 2n - 1 nodes.

Code Block 5.

```
def node.assign(i, v):
    if node.i <= i < node.j:
        if node.is_leaf():
            node.m = v
        else:
            node.l.assign(i, v)
            node.r.assign(i, v)

            # update the value at this node
            node.m = min(node.l.m, node.r.m)
```

Exercise 6. Show that, despite there being two recursive calls to .assign, this runs in $\mathcal{O}(\log n)$ time.

The `assign` operation updates a single leaf and then propagates the change upward.

At each node on the path from the root to the target leaf, the algorithm makes two recursive calls: one to the left child and one to the right child.

However, only one of these recursive calls continues further down the tree, the one containing the target index. The other child's subtree is completely skipped because the condition `node.i <= i < node.j fails for that child, so its recursive call returns immediately without recursing deeper.

Thus, the recursion depth is the height of the tree, O(log n), and at each level only O(1) work i done, with two recursive calls, but one terminates instantly. Total time is O(log n).

Exercise 8. Show that [*i*,*j*) contains [*i′*,*j′*) if and only if *i*≤*i′* and *j′*≤*j*.

By definition:
Interval [i,j) = {x∈Z | i≤x<j}
Interval [i′,j′) = {x∈Z | i’≤x<j’}

[i, j) contains [i’, j’) means every element of [i’, j’) is also an element of [i, j).

This holds if and only if:

1. The start of [i’, j’) is at least the start of [i, j): i <= i’
2. The end of [i’, j’) is at most the end of [i, j): j’ <= j

Both conditions are necessary and sufficient. Thus, [i, j) contain [i’,j’) if and only if i <= i’ and j’ <= j.

Exercise 9. Show that [*i*,*j*) is disjoint from [*i′*,*j′*) if and only if *j*≤*i′* or *j′*≤*i*.

Intervals [i, j) and [i’, j’) are disjoint if they have no element in common.
- [i, j) contains elements x such that i <= x < j
- [i’, j’) contains elements x such that i’ <= x j’

For them to have no overlap, all elements of the first must be before all elements of the second, or vice versa:

1. First interval ends before second starts:

j <= i’ (since intervals are half-open, j = i’ means no shared element).

2. second interval ends before first starts:

j’<= i

These are the only two ways for the intervals to be completely separate. Thus, [i, j) and [i’, j’) are disjoint iff j <= i’ or j’ <= i.

Exercise 10. If x_1 and x_2 are two distinct nodes on the same level representing intervals $[i_1, j_1)$ and $[i_2, j_2)$ respectively, then $[i_1, j_1)$ and $[i_2, j_2)$ are disjoint. Furthermore, if x_1 appears to the left of x_2, then $j_1 <= i_2$.

In a segment tree, each level partitions the array into disjoint intervals. Nodes at the same level represent intervals that do not overlap and together cover the array in order.

Thus, for two distinct nodes x_1 and x_2 on the same level:

$$[i_1, j_1) \cap [i_2, j_2) = \varnothing$$

If x_1 is to the left of x_2 in the tree by inorder traversal,, then its interval comes entirely before x_2's interval:

$j_1 <= i_2$ (the end of the left interval is before or at the start of the right interval)

This follows from the way the tree recursively splits intervals without gaps.

Exercise 11. Consider any interval [*i*,*j*) partially overlaps with the intervals of at most two nodes in each level of the segment tree.

At any given level of a segment tree, the intervals of nodes partition the array into disjoint, consecutive segments. For a query interval [i, j):

- Moving left to right across the level, the first node whose interval overlaps [i, j) may be partially overlapped on its right side.
- The next few nodes may be completely contained in [i, j).
- Eventually, we reach a node where [i, j) ends inside it, giving another partial overlap on its left side.

since intervals at the same level are disjoint and cover the array consecutively, [i, j) can partially overlap at most two nodes in that level:

- The first node it touches (if [i, j) starts inside it)
- The last node it touches (if [i, j) ends inside it)

All nodes in between are either completely contained or completely outside, never partially overlapped. Thus, at most two nodes per level have a partial overlap.

Exercise 12. Show that, despite there being two recursive calls to .min_range, this runs in O(log$n$) time. Hint: Recursion only continues every time there's a partial overlap; otherwise, it is handled in (1).

The min_range function recurses only on nodes whose interval partially overlaps with the query [i, j).

- From Exercise 11, at each level of the segment tree, at most two nodes have a partial overlap.
- All other nodes at that level are either completely contained (return in O(1)) or completely outside (return in O(1)).

Thus, the recursion only follows at most two paths down the tree, one for each boundary of the query interval.

- The tree height is O(log n).
- Each recursive call does O(1) work, checking conditions and combining results.

Therefore, total time is O(log n).

Exercise 13. Where is associativity used? What happens if you do it in a non-associative operation?

In a segment tree, the query combines results from multiple disjoint intervals (e.g., min(node.l.min_range(...), node.r.min_range(...))). The order of combining these results does not matter only if the operation is associative, because the final answer is built by combining partial results in arbitrary order as recursion unwinds.

If the operation is non-associative, then the result becomes order-dependent. since the recursion order is fixed (left then right), the query would still produce a consistent result for that order, but it would **not correctly** represent the intended operation over the whole range if the operation is meant to be applied in a specific order (e.g., subtraction or division).

Thus, segment trees assume associativity to guarantee correctness regardless of combination order.

Exercise 14. Show that segment trees can still be used for an associative operator that doesn't have an identity element. Hint: Invent an identity element.

Even if the associative operator $\oplus$ has no natural identity, we can invent an identity element e that behaves as one for our purposes:

- Define e such that for any x, e $\oplus$ x = x $\oplus$ e = x.

- If the operator has no such element in the original domain, we can extend the domain by adding a special value e with this property.

In a segment tree query, we need an identity only for nodes that are completely non-overlapping, to return a neutral value that doesn't affect the combination. By adding e artificially, we ensure:
- If a node's interval is disjoint from the query, we return e.
- Combining any result with e leaves it unchanged.

Thus, the query still works correctly without needing the original domain to contain an identity.

Exercise 15. Adapt our segment tree solution for Problem 1 to also return the index that achieves min_range(i,j), i.e., the indexk $\in$ [i, j) such that a_k is the minimum among a_i, a_{i+1}, …, a_{j-1}.

We have each node store both the minimum value in its interval and the index where that minimum occurs.

```
class Node:
    i, j # interval [i, j)
    min_val # minval of this interval
    min_idx # index of that minimum
    left, right # children
```

```
so for each node, it would look like:
For a leaf: min_val = a[i], min_idx = i
For an internal node:
    Compare left.min_val and right.min_val:
        If left.min_val <= right.min_val:
            min_val = left.min_val, min_idx = left.min_idx
        Else:
            min_val = right.min_val, min_idx = right.min_idx
```

Query min_range(i, j):
- If completely contained: return (node.min_val, node.min_idx)
- If disjoint: return (∞, -1).
- Else: recursively query children, then combine as in building (choose smaller value, keeping its index).

This returns the index of the minimum in O(log n) time.

Exercise 17. Can you use the postorder traversal for this technique (flattening trees for subqueries)?

No, postorder traversal does **not** work for this technique.

In postorder traversal, a node is visited after all nodes in its subtree. This means the subtree nodes are **not contiguous** in the traversal order, they are split into left subtree, right subtree, and then the node itself, breaking contiguity.

We need each subtree to correspond to a single contiguous range in the flattened array.

Preorder works because it visits a node first, then recursively traverses its children.
Postorder fails because the node appears at the end of its subtree traversal, and the subtree nodes are not together as one block.

Thus, for subtree -> range queries, **preorder** (or a similar order where parent comes before children) is required.

Exercise 18. Show how to augment the DFS traversal so that the interval [*i*,*j*) corresponding to each node can also be computed. The DFS traversal should still run in $\mathcal{O}(n)$ time.

We augment DFS to assign entry time and exit time for each node:

```
time = 0
def dfs(u):
    entry[u] = time
    time += 1
    arr.append(value[u]) # flattened array
    for v in children[u]:
        dfs(v)
    exit[u] = time - 1 # or exit[u] = time if using [entry, exit)
```

The subtree of node u corresponds to interval [entry[u], exit[u]] in flattened array, and is still O(n) time, each node visited once doing O(1) work.

Exercise 20. show that if the tree is not a line, then it is impossible to flatten the nodes of a tree into a sequence such that every path becomes a range. (Hint: If the tree is not a line, then there is a node of degree >= 3).

Consider three distinct neighbors *a,b,c* of v. In any flattening into a sequence, these four nodes appear in some order.

- A path in the tree can include any two of these neighbors via v (e.g., a -> v -> b).
- For this path to be a contiguous range in the sequence, *a* and *b* must appear consecutively with v between them in the order.
- But the same must hold for a->v->c and b->v->c.

This is impossible in a linear order:

- If v is between a and b, and also between a and c, then b and c must be on the same side of v.
- Then path b->v->c would have b and c on the same side of v in the sequence, so v is not between them, breaking contiguity.

Thus, no linear ordering can make every path a contiguous range in a non-line tree.

Exercise 21. Let x be a non-root node, and let p be its parent.

Let weight(x) = number of nodes in subtree rooted at x.

1. **Prove that weight(x) < weight(p).**

Then x is a child of p, so subtree of x is strictly contained in subtree of p, since p itself is also in its subtree but not in x's subtree. Therefore, weight(x) <= weight(p) - 1 < weight(p).

2. **Prove that if (p, x) is a light edge, then 2 * weight(x) < weight(p).**

By definition, a light edge (p, x) means x is not a child with maximum weight. Let y be the heavy child of p (child with largest subtree). Then:

$$\text{weight}(y) >= \text{weight}(x)$$

and since x != y:

$$\text{weight}(y) > \text{weight}(x)$$

subtree of p includes p itself, subtree of x, subtree of y, and possibly other children, so:

$$\text{weight}(p) >= 1 + \text{weight}(x) + \text{weight}(y) + \dots$$

since weight(y) > weight(x):

$$\text{weight}(p) > 1 + \text{weight}(x) + \text{weight}(x) = 1 + 2\text{weight}(x)$$

For integer weights, weight(p) >= 2 * weight(x) + 1 > 2 * weight(x).

Thus, 2 * weight(x) < weight(p).

Exercise 22. Use Exercise 21 to prove that there are <= lg n light edges on the path from any node to the root.

Let the path from node x to the root be $x = v_0, v_1, v_2, \ldots, v_k = \text{root}$.
Consider the weights along this path:

$$w_0 = \text{weight}(v_0),\ w_1 = \text{weight}(v_1),\ \ldots,\ w_k = \text{weight}(v_k) = n$$

By Exercise 21.1, weights strictly increase: $w_0 < w_1 < \ldots < w_k = n$. When traversing a light edge (v_{i-1}, v_i), Exercise 21.2 gives:

$$2 * w_{i-1} < w_i$$

Let m be the number of light edges on the path, start from $w_0 >= 1$. Each light edge at least doubles the weight (since $w_i > 2w_{i-1}$). Heavy edges increase weight but by a smaller factor.

Thus, after m light edges:

$$n = w_k >= 2^m * w_0 >= 2^m * 1 = 2^m$$

Taking $\log_2$:

$$m <= \log_2 n$$

Therefore, there are at most floor($\log_2 n$) light edges on any root-to-node path.

Exercise 23. Prove that for any two nodes i and j, there are <= 2 lg n light edges on the path from i to j.

- **Hint: The path i ⇝ j can be split up into i ⇝ lca(i,j) and lca(i,j) ⇝ j.**
- **Hint 2: Note that i → lca(i,j) is a subpath of the path i → root.**

Let l = lca(i,j).

The path i ⇝ j consists of:

- upward path: i ⇝ l
- downward path: l ⇝ j

Upward path: i ⇝ l is a subpath of i ⇝ root.
By Exercise 22, i ⇝ root has <= log n light edges. so i ⇝ l, also has <= log n light edges, by subpath property.

Downward path: l ⇝ j is the reverse of j ⇝ l, which is a subpath of j ⇝ root. Using the same reasoning, there are <= log n light edges on j ⇝ l, hence also on l ⇝ j. Light edges on the two parts are disjoint, they occur in different directions, but light edge definition is symmetric for counting.

so total <= logn + logn = 2logn

Thus, path i ⇝ j contains <= 2 logn light edges.

Exercise 24. show that if i.topmost is deeper than j.topmost, then i is not on the same preferred path as lca(i, j).

Let l = lca(i,j)

Given: topmost[i] is deeper than topmost[j], meaining depth(topmost[i]) > depth(topmost[j]).

suppose for contradiction, that i is on the same preferred path as l. Then, l is an ancestor of i on that preferred path. By definition, topmost[i] is the highest node on that preferred path. since l is on the same path and above i, we have topmost[i] is an ancestor of l (or equal to l if l i the topmost).

Thus, depth(topmost[i]) <= depth(l).

But l is an ancestor of j, so depth(l) <= depth(topmost[j]), since topmost[j] is some node on the path from j to root, possibly above l.

Combining:

depth(topmost[i]) <= depth(l) <= depth(topmost[j])

This contradicts depth(topmost[i]) > depth(topmost[j]). Therefore, i cannot be on the same preferred path as l.

Therefore, we can simply compare the depths of i.topmost and j.topmost, and perform a climb on whichever's deeper. If the depths are equal, but they aren't on the same preferred path, then we can perform a climb on either since either choice progresses toward the root without overshooting LCA.

Exercise 25. Explain how to solve LCA queries online with $\mathcal{O}(n)$ preprocessing and $\mathcal{O}(\log n)$ per query using heavy path decomposition.

We first do O(n) preprocessing by:

1. Running DFS to compute:
    - parent[u], depth[u]
    - size[u] (size of the subtree)
    - heavy[u] = child with max size (or -1 if leaf)
2. Run a second DFS to assign head[u] (top of every path):
    - start new path at root, traverse heavy child first.
    - for heavy child: head[heavy] = head[u]
    - for light child: head[child] = child (this starts a new path)
3. Compute pos[u] for array representation in RMQ

We now do LCA queries in O(log n)

```
def lca(u,v):
    while head[u] != head[v]:
        if depth[head[u]] > depth[head[v]]:
            u = parent[head[u]]
        else:
            v = parent[head[v]]
    # now same heavy path
    return u if depth[u] < depth[v] else v
```

This runs in O(log n) because:

- each iteration moves one node to the parent of its path head.
- this crosses at least one light edge.
- by exercise 22, there are <= log n light edges root-to-node.
- so, there are <= 2 log n = O(log n) iterations in total

Thus, LCA is O(log n) with O(n) preprocessing.