Problem 1: Min and Max

Given an array `a[0..n]` of `n` positive integers build a data structure to answer two different types of queries:

1. `update(l, r, t)` that updates every `a[i]` with `min(a[i], t)`, where `l <= i <= r`
2. `max(l, r)` that returns the largest value in `a[l..r]`
   A sequence of `m` queries is provided and the target solution must run in $O((n+m)\log(n))$ time.

To solve the problem it is provided an implementation of a Segment Tree.

Segment Tree Implementation

Each `Node` `n` of the segment tree contains:

- `key: u32`, the maximum of the segment covered by `n`
- `left_edge: usize`, the left edge of the segment covered by `n`
- `right_edge: usize`, the right edge of the segment covered by `n`
- `left_child: Option<Box<Node>>`, the left subtree rooted in `n`
- `right_child: Option<Box<Node>>`, the right subtree rooted in `n`

The tree is represented as a single `Node` root.

Construction of the Tree

The segment tree is built recursively, passing:

- the array `a` received in input
- the current node `curr_node`
- `0` and `a.len()-1` as the first segment edges

If `l == r` it means that we reached a leaf, and hence we set `curr_node.key = a[l]`. Otherwise, the following operations are done:

- compute `m`, the half of the current node segment
- build a new node `left_child` that covers the segment `[l, m]`
- recursively build the left subtree, using `left_child` as `curr_node`
- build a new node `right_child` that covers the segment `[m+1, r]`
- recursively build the right subtree, using `right_child` as `curr_node`
- compare the `right_child.key` and `left_child.key` and assign the maximum as key to `curr_node`
- add `right_child` and `left_child` as children of `curr_node`

Max

The first kind of query is of the form `max(l, r)` that returns the largest value in `a[l..r]`.

The function

```
fn max_rec(curr_node: &Node, l: usize, r: usize) -> u32
```

has as parameters:

- `curr_node`, the current root of the tree
- `l`, the left endpoint of the queried segment
- `r`, the right endpoint of the queried segment

and it recursively computes the maximum in `[l,r]`.

The function behaves as follows:

- checks that `l <= r`
- if `curr_node.left_edge == curr_node.right_edge` means that the current node is a leaf, hence returning the current node's key
- if the `curr_node` segment covers exactly the queried one `[l,r]`, hence return the current node's key
- if the queried segment is fully contained on either the `left_child` or `right_child` return the result recursively computed on it
- if the queried segment spans on both the left and right subtree of the current node we compute the middle element `m` and compute the maximum from the left and right by recurring on `[l, m]` and `[m+1, r]` and return the greatest between the two.

Time Complexity: $O(\log(n))$

- the function traverses the segment tree from the root to the leaves, and for each node, it performs constant-time operations
- the height of the segment tree is $\log(n)$, as it is a binary tree

Space Complexity: $O(n)$

- the segment tree is represented using a binary tree structure with additional information for each node, the space required is proportional to the number of nodes in the tree

Update

The second kind of query is of the form `update(l, r, t)` that replaces every value `a[i]` with `min(a[i], t)`, where `l <= i <= r`.

The function

```
fn update_rec(curr_node: &mut Node, l: usize, r: usize, t: u32)
```

has as parameters:

- `curr_node`, the current root of the tree
- `l`, the left endpoint of the queried segment
- `r`, the right endpoint of the queried segment

and it recursively updates the values in `a[l..r]`.

The function behaves as follows:

- if the current node segment is outside the queried update simply return
- if the queried segment is fully inside the current node's segment, update the `curr_node.key` and propagate the update on the left and right subtree recurring on `left_child` and `right_child` and then return
- recur on both the left and right subtrees of the current node and update the current node's key, which might have changed due to the update

Problem 2: Is There

Given `n` segments `[l,r]` such that $0 \leq l \leq r \leq n-1$ the task is to solve `m` queries `is_there(i,j,k)`, where `is_there(i,j,k)` has to return 1 if there exists a position p with $0 \leq i \leq p \leq j \leq n-1$.

The solution exploits two techniques:

- prefix sum
- binary search

The function

```
fn is_there(segments: &Vec<(u32, u32)>, query: (u32, u32, u32)) -> bool
```

has as parameters:

- `segments`, the array of segments `[l,r]` received in input
- `query`, a single triple, where the first element is `i`, the second is `j` and the last is `k`

and it returns `true` if such a position is found.

The function behaves as follows:

- create an array `axis` of length `segments.len()+1`, initialized to all zeroes

- mark the array `axis` so that each element of it represents the number of "active" segments that start in `i`
    - for each `segment` in `segments`
        - `axis[segment.0]+=1`
        - `axis[segment.1+1]-=1`
- compute the number of active segments for every position, this is done by computing the prefix sum of `axis`
- create a map that associates the number of active segments on a position `i` to the "list" of positions that have that number of active segments
- exploit binary search:
    - to the binary search pass the "list" of positions that have `k` active segments, retrieved through the map
    - search the "list" of positions, returning `true` as soon as a position that is in `[i,j]` is found.
    - if such a position is not found return `false`

Time and Space Complexity are analogous to the previous problem.