Hands-on 2 Solution Tommaso Crocetti

Node structure

The implementation of the segment trees for the two problems uses the node structure presented on the course's website, also modified to contain the left and right limits of the corresponding interval. The key values of the node used for problem #1 are of type u32, and represent the maximum value in their interval. They also include a lazy field, used for the lazy upgrade needed for the range update. The default value is u32::MAX since the update requires computing the minimum between the keys in the subrange and the parameter k.

Node Minandmax

```
pub struct Node {
    key: u32,
    left: usize,
    right: usize,
    id_left: Option<usize>,
    id_right: Option<usize>,
    lazy: u32
}
```

The node used in problem #2 maintains a HashMap used to check if, in the current interval, there is a point contained by exactly k segments. The map for a given node maps the different cardinality of the overlaps in the subrange (so considering position p such that l<= p <=r) with how many times they are present. The lazy field isn't necessary since the collection is never updated.

Node Isthere

```
pub struct Node {
    key: HashMap<i32, u32>,
    left: usize,
    right: usize,
    id_left: Option<usize>,
    id_right: Option<usize>,
}
```

Segment tree construction

The segment tree for the two problems is just a vector of Nodes, built starting from the array given in the input of the function **new**, which creates an instance of the segment tree with an empty vector, calls the **build** method, and then returns the just built tree. For problem #1, the array given and used is simply the input array containing u32 values, but problem #2 needs some preprocessing: at first, the given array contains the n segments (represented as couples of u32), but for the requested query we are interested only in the number of overlaps for every position. The function **get_prefix** first creates a new array (propagate) of size n, then for every segment (l, r) it increases by 1 propagate[l] (a new segment starts) and decreases by one propagate[r + 1] (a segment ended). Applying prefix sum on this array gives the number of overlaps for each position.

New Minandmax

```
    pub fn new(arr: &[u32]) -> Self {
        let mut tree = Minandmax { nodes: Vec::new() };
        tree.build(0, arr.len() - 1, arr);
        tree
    }
```

New Isthere

```
    pub fn new(arr: &[(u32, u32)]) -> Self {
        let mut tree = Isthere { nodes: Vec::new() };
        let prefix = get_prefix(arr);
        tree.build(0, arr.len() - 1, &prefix);
        tree
    }
```

Function get_prefix

```
fn get_prefix(arr: &[(u32, u32)]) -> Vec<i32> {
    let n = arr.len() as usize;
    let mut prefix: Vec<i32> = vec![0; n];
    for i in arr {
        prefix[i.0 as usize] += 1;
        if ((i.1 + 1) as usize) < n {
            prefix[(i.1 + 1) as usize] -= 1;
        }
    }
    for i in 1..n {
        prefix[i] += prefix[i - 1];
```

```
    }
    prefix
}
```

The **build** methods recursively build the segment tree with a strategy similar to a post-ordered visit, starting from an interval (0, arr.len() – 1). The case base is when a leaf is encountered (l == r), in this case, the method creates a new Node with the value of arr[l] for problem #1, and with a HashMap containing (arr[l], 1) for problem #2. In a recursive call, the method calls on the left and right subinterval by splitting in half the current interval and then creating a new node. The recursive calls return the indexes of the two children nodes created so that the current node can point to them with id_left and id_right. For problem #1, a non-leaf node has a key corresponding to the maximum key in the two children node, while for problem #2 the Hashmap is the merge between the two children HashMap.

Build Minandmax

```
    fn build(&mut self, l: usize, r: usize, arr: &[u32]) -> usize {
        if l == r {
            let node = Node::new(arr[l], l, r);
            self.node.push(node);
            return self.nodes.len() - 1;
        }
        let mid = (l + r) / 2;
        let id_left = self.build(l, mid, arr);
        let id_right = self.build(mid + 1, r, arr);
        let key = self.nodes[id_left].key.max(self.nodes[id_right].key);
        let mut node = Node::new(key, l, r);
        node.id_left = Some(id_left);
        node.id_right = Some(id_right);

        self.nodes.push(node);
        self.nodes.len() - 1
    }
```

The time required to create a segment tree for problem #1 depends only on the number of nodes created since the case base and the combination takes constant time. Given an array of size n, the number of nodes in the segment tree is exactly 2n-1:

1. Base case, n = 1, clearly the number of nodes in the segment tree is N = 1, and N == 2n-1
2. Inductive case, suppose that for an array of size n its segment tree has size N=2n-1. Then adding one element in the array would modify the interval size of a node in the tree, particularly a leaf node. Now that node has to point to the new leaf. So with n'=n+1 elements in the array, we increase the size of the segment trees by 2, giving N'=N+2 = 2n+1, which is equal to 2n'-1 (= 2(n+1)-1 = 2n+1)

The time and space complexity required for the **build** method is O(n)

Build Isthere

```
    fn build(&mut self, l: usize, r: usize, arr: &[i32]) -> usize {
        if l == r {
            let mut map = HashMap::new();
            map.insert(arr[l], 1);
            let node = Node::new(map, l, r);
            self.nodes.push(node);
            return self.nodes.len() - 1;
        }
        let mid = (l + r) / 2;
        let id_left = self.build(l, mid, arr);
        let id_right = self.build(mid + 1, r, arr);
        let mut map = HashMap::new();
        for (&k, &v) in &self.nodes[id_left].key {
            *map.entry(k).or_insert(0) += v;
        }
        for (&k, &v) in &self.nodes[id_right].key {
            *map.entry(k).or_insert(0) += v;
        }
        let mut node = Node::new(map, l, r);
        node.id_left = Some(id_left);
        node.id_right = Some(id_right);
        self.nodes.push(node);
        self.nodes.len() - 1
    }
```

The same reasoning is also valid for this **build** method, but the time required to build a non-leaf node is no longer constant. Time and space complexity so depend on the number of insertions for each node, for each level of the tree. Considering a completely balanced tree with n leaf, for each leaf we have only

one insertion, so the total time and space used is O(1) * n = O(n). The level above contains n/2 nodes, and for each of them we have 2 operations (2 different insertions or 1 insertion and 1 update), so we spend O(1) * n/2 = O(n). This process can be applied with the same result till the root, where we have to consider just one node, but the number of operations required is O(n). Since the height of the tree is logarithmic, the total time and space complexity is O(nlog(n))

Problem #1 – Minandmax

The first problem requires range updates, which modifies the values in the array to the minimum between the given and the current values, and queries, that outputs the maximum value in the subrange. The function **propagate** executes lazy propagation of the updates for the current node. It updates the value of the current node to the minimum between its key and its lazy value, then sets the lazy value of the node's children to the minimum between the old and new lazy values. In the end, it reset the lazy field to u32::MAX. The update is correct because, for a leaf, it just sets the key to the minimum value between the two numbers analyzed, for an internal node of the segment tree (in case of a total overlap) it modifies in the same way: the maximum value is forced to be the minimum between the node's key and the lazy value since the update will decrease every value less or equal to the maximum but greater than the lazy value. So it's correct to update in this way also the internal node values since there can't be values less than the maximum but greater than the lazy value.

Function propagate

```
    fn propagate(&mut self, id: usize) {
        self.nodes[id].key = self.nodes[id].key.min(self.nodes[id].lazy);
        if let Some(left) = self.nodes[id].id_left {
                                                self.nodes[left].lazy      =
self.nodes[left].lazy.min(self.nodes[id].lazy);
        }
        if let Some(right) = self.nodes[id].id_right {
                                               self.nodes[right].lazy      =
self.nodes[right].lazy.min(self.nodes[id].lazy);
        }
        self.nodes[id].lazy = u32::MAX;
     }
```

All operations take constant time, so it requires O(1) time complexity.

The function **range_update** performs the range update with lazy propagation. Given the current node's id, the range, and the value requested, first propagate

the updates if it's pending, then it checks the possible overlap cases between the current node's interval and the requested interval:

1. No overlap: just return.
2. Total overlap: update the lazy value of the current node to the minimum between the previous lazy value and the given value, then propagate the update instantly.
3. Partial overlap: perform a **range_update** on both children if they exist, then update the current node value so that it contains the query solution for its interval, setting the key to the maximum between the two children's keys.

Function range_update

```
    fn range_update(&mut self, node_id: usize, ql: usize, qr: usize, value:
u32) {
        self.propagate(node_id);
        if qr < self.nodes[node_id].left || ql > self.nodes[node_id].right
{
            return;
        }
        if ql <= self.nodes[node_id].left && self.nodes[node_id].right <=
qr {
            self.nodes[node_id].lazy = value;
            self.propagate(node_id);
            return;
        }
        if let Some(left) = self.nodes[node_id].id_left {
            self.range_update(left, ql, qr, value);
        }
        if let Some(right) = self.nodes[node_id].id_right {
            self.range_update(right, ql, qr, value);
        }
                                        self.nodes[node_id].key         =
self.nodes[self.nodes[node_id].id_left.unwrap()].key.max(self.nodes[self.
nodes[node_id].id_right.unwrap()].key);
    }
```

Time complexity depends on the number of iterations and their single cost. Since every operation takes constant time, even the propagation, the complexity depends only on the number of iterations. Since it follows the same visit of the tree as a query, the time complexity is O(log(n)).

The function **query** returns the maximum value in the required subrange. It first propagates the lazy update of the current node if pending, then checks the overlap:

1. No overlap: just return None (no interest in the subrange).
2. Total overlap: just return the solution for the subrange, represented by the current node's key.
3. Partial overlap: check the recursive solutions for the node's children, and return the maximum between the two.

Function query

```
    fn query(&mut self, node_id: usize, ql: usize, qr: usize) -> Option<u32>
{
        self.propagate(node_id);

        // Caso 1: Intervallo completamente fuori
        if qr < self.nodes[node_id].left || ql > self.nodes[node_id].right
{
            return None;
        }

        // Caso 2: Intervallo completamente dentro
        if ql <= self.nodes[node_id].left && self.nodes[node_id].right <=
qr {
            return Some(self.nodes[node_id].key);
        }

        // Caso 3: Intervallo parzialmente dentro
        let mut max = 0;
        if let Some(left) = self.nodes[node_id].id_left {
            if let Some(lres) = self.query(left, ql, qr) {
                max = max.max(lres);
            }
        }
        if let Some(right) = self.nodes[node_id].id_right {
            if let Some(rres) = self.query(right, ql, qr) {
                max = max.max(rres);
            }
        }
        Some(max)
    }
```

It follows the typical query structure for a segment tree, every operation in a call takes constant time, so its time complexity is O(log(n)).

Finally, the **update** and **max** functions just support that call respectively, the **range_update** and **query** methods, adding the first node_id to start the execution as the first parameter, the id of the root. Since the construction of a segment tree happens with a post-order visit, the root is always the last node in the vector.

Function update

```
    pub fn update(&mut self, l: usize, r: usize, value: u32) {
        let root_id = self.nodes.len() - 1;
        self.range_update(root_id, l, r, value);
    }
```

Function max

```
    pub fn max(&mut self, l: usize, r: usize) -> Option<u32> {
        let root_id = self.nodes.len() - 1;
        return self.query(root_id, l, r);
    }
```

Problem #2 – IsThere

The second problem requires only queries that return if there exists in the required subrange a position p such that exactly k segments overlap in there. The query functions use the result computed and stored in each node's HashMap to build the answer, as before checking the overlap between the requested interval and the current interval:

1. No overlap: just return false (no interest in the subrange).
2. Total overlap: check if the current node's HashMap contains an entry with the desired value, in case return true, otherwise false.
3. Partial overlap: check the recursive solutions for the node's children, and return the or between them (it's needed at least one position).

Function query

```
    fn query(&self, node_id: usize, i: usize, j: usize, k: i32) -> bool {
        let node = &self.nodes[node_id];

        // Caso base: l'intervallo del nodo non interseca [i, j]
        if node.right < i || node.left > j {
```

```
            return false; // Non contribuisce
        }

        // Caso base: l'intervallo del nodo è completamente dentro [i, j]
        if i <= node.left && node.right <= j {
            // Controlla se il valore `k` è presente nel nodo corrente
            return node.key.get(&k).copied().unwrap_or(0) > 0;
        }

        // Caso generale: dividi tra figlio sinistro e destro
        let mut left_result = false;
        let mut right_result = false;

        if let Some(left_id) = node.id_left {
            left_result = self.query(left_id, i, j, k);
        }

        if let Some(right_id) = node.id_right {
            right_result = self.query(right_id, i, j, k);
        }

        left_result || right_result // Ritorna `true` se almeno un figlio
restituisce `true`
    }
```

Since we can assume that a search query in a HashMap takes constant time, the time complexity is O(log(n)).

As before, the function isthere just calls the query function passing all the parameters, adding first the id of the root.

```
    pub fn isthere(&self, i: usize, j: usize, k: i32) -> u32 {
        let root_id = self.nodes.len() - 1;
        return self.query(root_id, i, j, k) as u32;
    }
```

Main

The main functions for both problems (respectively min_and_max.rs and is_there.rs) read the input from the given file (it must be passed with clap, see the Usage section). The file must respect the structure specified in the hands-on website.

For problem #1, it reads the number of elements in the array (n) and the number of queries (m) from the first line, then fills the array with the value in the second line, so to create the segment trees. For all the remaining lines, it checks if it is a query on an update request and performs it.

For problem #2 it's used the same approach. The only difference is that the array is built reading from n different lines, where each line is processed as a u32 couple, and the requests are all queries.

Usage

The shared zip file contains the shown code in the following structure:

- src: contains all the code, grouped as:
  - lib: a folder containing the two implementations of the segment trees.
  - bin: a folder containing the two main files for the two problems.
  - lib.rs: file that makes accessible the two libraries in lib.
- test_1: a folder containing input and output given for problem #1.
- test_2: a folder containing input and output given for problem #2.

After applying cargo build, the command to run one of the two main file is:

cargo run --bin <main file> -- --file .\test_<1 || 2>\input<input_number>.txt