Segment Tree Proof using Dafny 3.13

What was done:

1. Implemented Monoid interface with 2 examples (minimum over bounded integers and sum over integers) in Dafny
2. Implemented Segment Tree using Dafny and verified it
3. Created an example of usage in Main method
4. Confirmed that Monoid condition is enough for template Segment Tree

Part 1. Monoids

We define monoid as a trait with 2 function methods:
identity() - returns neutral element
op(x, y) - returns result of monoid operation
Any monoid used in segment tree needs to have validMonoid() predicate, which states that monoid is associative and identity() satisfies neutral element properties under op
We also provide 2 classes: minimum over bounded integers (constant 100 is used here) and sum over integers. Dafny does not require additional lemmas to verify that both of them are valid monoids.
Note that we are assuming that template parameter for Monoid is reference-free, which is a valid assumption for most of the cases where we want to use segment tree (as we aim for logn queries, but storing arrays or sets with unbounded size is not what we expected for every of O(n) elements of segment tree).

Part 2. Segment Tree structure

Final decision is to use a functional approach for building the tree. Although normally in Competitive Programming code tree is built more explicitly with heavy reliance on the powers of two, it is not that beneficial for verification, as many non-trivial properties of powers of two need to be proven and carefully used. Other popular in imperative languages pointer-based approach was discarded due to problems with reading and modifying clauses, as it is hard to allow the tree to read it's left child and left child's left child.
So we define our datatype as:

```
datatype SegmentTree<T> =
   Leaf(m: int, value: T)
   | Node(l: int, m: int, r: int, left: SegmentTree, right: SegmentTree, value: T)
```

For Leaf m corresponds to the unique index in array. For Node l and r are boundaries of the root node, left and right are children trees, and m is a split parameter which will be used later. Also Leaf and Node both store some value.

Part 3. Components

Below is a brief overview of the proof.

We provide a definition for valid structure: that is, Leaf always has valid structure, and Node has valid structure if children are valid and their root segments are [l, m], [m + 1, r] respectively.
Then we provide a definition for straightQuery over a subarray of some array and some monoid - essentially it is just a right fold on the corresponding elements.
We then prove that for associative monoid right fold on tangent segments is combination of right folds of each segment in lemma associativeQuery.
Then we define a recursive buildTree function method which builds the valid tree on some subsegment.
Then we create an interface for quick query answering using innerQuery. It calculates the answer on the intersection of tree interval and [l, r] recursively.
Then we create an interface for quick change of a single element. It uses rebuildTree - function method which returns updated tree, where node has been changed only if its interval contained the updated position.

Part 4. Main

In the Main function you can see examples of usage for segment trees and monoids for two most popular examples from competitive programming tasks: addition and minimum over a segment.

Possible further improvements and future work

- It seems that the lazy segment tree could be verified as well, but the invariants are more complicated and we need an additional wrapper over monoid with a second template argument and 2

more functions. This was the primary goal which eventually was not achieved.

- As we used a functional style datatype for tree and most of the functions rely on recursion rather than cycles, it means that the Dafny code can be potentially used for segment tree implementation on pure functional languages like Haskell, which is interesting.
- It seems possible to analyze time complexity as well - under the assumption that we able to copy immutable objects in constant time, we can calculate number of recursive calls in innerQuery and rebuildTree to conclude that we made O(logn) calls at maximum