HandsOn 1

Studente: Luca Rizzo
Matricola: 598992

Exercise 1

The goal of the first exercise was to verify whether a given binary tree satisfies the Binary Search Tree (BST) property. The original *Tree* definition was slightly modified to allow negative keys. The solution is implemented in the function *is_bst*, defined in the corresponding module. It relies on the auxiliary function *rec_is_bst*, which performs an in-order traversal of the tree and applies a short-circuit mechanism whenever the property is violated for a node. Unlike the approach presented during the lecture (which used a post-order traversal and gathered information from child nodes to check BST property) this implementation follows a top-down strategy, where each child receives from its parent the range of allowed values for its key **with exclusive bounds**, based on the nodes already visited. More precisely:

- When recursing on the left subtree, the allowed range is updated by keeping the same minimum and replacing the maximum bound with the current key.
- When recursing on the right subtree, the range is updated by keeping the same maximum and replacing the minimum bound with the current key

A *None* node trivially satisfies the property. The range bounds are represented using *Option<i32>* instead of fixed constants (*i32::MIN, i32::MAX*). This design choice allows the inclusion of extreme integer values and ensures that a tree such as *[2, i32::MIN, i32::MAX]* is considered valid (as verified in the test *test_depth_one_tree_min_max*). Since the algorithm performs a single traversal of the tree, the time complexity is *O(n)*, where *n* is the number of nodes, and the auxiliary space is proportional to the recursion depth, i.e., *O(h)* where *h* is the height of the tree.

Exercise 2

The goal of the second exercise was to identify the maximum path sum between two leaves of a binary tree. As in the previous task, the *Tree* definition was modified to allow negative keys. The solution is implemented in the function *max_path_sum*, which relies on the auxiliary function *rec_max_path_sum* to perform a post-order traversal of the tree.
The core idea is to apply a divide-and-conquer approach: each recursive call computes results for the left and right subtrees, and the current node combines them to reconstruct the optimal solution. Specifically, the recursive function returns two values:

1. the maximum path sum between two leaves found so far within the subtree;
2. the best downward path, that is, the maximum sum of a path from any leaf to the current node, which can be used by the parent node to form longer paths.

For each node if the node has both left and right children, it means there are at least two leaves in its subtrees: therefore, the sum of the best downward paths from the left and right children plus the current node's key can form a new candidate for the global maximum path.
If the node does not have both children, it cannot form a valid leaf-to-leaf path passing through itself. In this case, it simply propagates upward the maximum value received from its child and the sum of that child's path plus its own key. For a *None* node, the function returns *i32::MIN* for both values. This sentinel value indicates the absence of a valid path between two leaves and ensures

that single-branch trees are correctly handled. In this implementation, `Option` is intentionally avoided to keep the logic simpler; as a result, the sentinel value may denote either a genuine minimal sum or the non-existence of a valid leaf-to-leaf path. Since the algorithm performs a single traversal of the tree, the time complexity is $O(n)$, where n is the number of nodes, and the auxiliary space is proportional to the recursion depth, i.e., $O(h)$ where h is the height of the tree.