# Complete Documentation for "House Robber III" (LeetCode 337)

The "House Robber III" problem introduces an interesting variation of the classic house robber problem by arranging houses in a binary tree structure. This setup adds a layer of complexity, as robbing a house (node) directly affects its children, prohibiting their robbery. The goal is to determine the maximum sum of money that can be stolen without triggering any alarms. The problem exemplifies a balance between greed and strategy, making it a compelling study in dynamic programming and recursive tree traversal.

The core intuition lies in exploring two possible states for each house: robbing or skipping it. Robbing a house implies that its direct children cannot be robbed, but skipping it leaves the decision for its children open. This binary decision-making process naturally forms overlapping subproblems, as the maximum amount robbed from a node depends on the maximum values obtained from its left and right subtrees. This recursive dependency allows us to optimize the solution using a postorder traversal of the binary tree.

The solution leverages a bottom-up approach, where the results for the subtrees are computed first and then combined for the parent node. For each node, we calculate two values: the amount robbed if the node itself is robbed, and the amount if it is skipped. These two values are recursively calculated for its left and right children and are used to determine the best decision for the current node. This localized optimization ensures that every subtree contributes optimally to the overall solution.

Dynamic programming plays a pivotal role in this solution, but unlike traditional table-based DP, here we use the tree's inherent structure to store intermediate results implicitly. The results are computed during recursion and passed back up the call stack, which eliminates the need for additional data structures. This approach minimizes space usage while ensuring optimal performance. Each node is visited exactly once, leading to an efficient O(n) time complexity, where n is the number of nodes in the tree.

Space complexity primarily depends on the recursion stack. In a balanced binary tree, the height of the tree is O(logn), leading to a logarithmic space requirement. However, in a skewed tree, the height can be O(n), resulting in linear space usage. Thus, the overall space complexity ranges from O(logn)to O(n) depending on the shape of the tree. These bounds ensure that the algorithm performs well even for large binary trees.

This problem highlights real-world applications where hierarchical decision-making under constraints is critical. Examples include managing hierarchical networks, resource allocation in organizational structures, or planning investments with dependency restrictions. The binary tree structure is an excellent representation for such problems, making this solution versatile for broader applications beyond binary trees.

The problem also emphasizes the importance of handling edge cases, such as trees with no nodes or only one node, which can be solved trivially. Moreover, while the solution works efficiently within the given constraints, it can be extended to handle more complex scenarios, such as trees with variable constraints or additional relationships between nodes. Such extensions could involve N-ary trees or dynamic rules, requiring further generalization of the current approach.

Overall, the "House Robber III" problem elegantly combines tree traversal and dynamic programming principles to solve a constrained optimization problem. Its recursive design ensures simplicity and clarity, while its efficiency makes it suitable for practical scenarios. The problem serves as an excellent example of how foundational algorithms can be adapted to solve complex, real-world challenges in hierarchical systems.