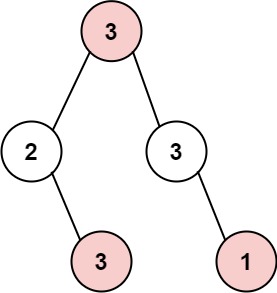
<https://leetcode.com/problems/house-robber-iii/description/>

The thief has found himself a new place for his thievery again. There is only one entrance to this area, called root.

Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that all houses in this place form a binary tree. It will automatically contact the police if **two directly-linked houses were broken into on the same night**.

Given the root of the binary tree, return *the maximum amount of money the thief can rob****without alerting the police***.

**Example 1:**

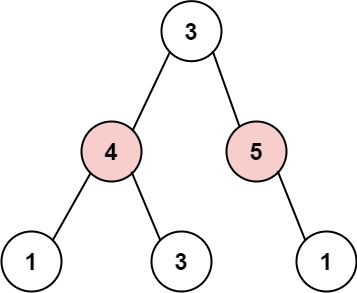


**Input:** root = [3,2,3,null,3,null,1]

**Output:** 7

**Explanation:** Maximum amount of money the thief can rob = 3 + 3 + 1 = 7.

**Example 2:**



**Input:** root = [3,4,5,1,3,null,1]

**Output:** 9

**Explanation:** Maximum amount of money the thief can rob = 4 + 5 = 9.

**Constraints:**

The number of nodes in the tree is in the range [1, 10^4].

0 <= Node.val <= 10^4

**Attempt 1: 2025-06-01**

**Solution 1: Native DFS (10min, TLE 122/124)**

class Solution {

    public int rob(TreeNode root) {

        return helper(root, false);

    }

    private int helper(TreeNode node, boolean parentRobbed) {

        if(node == null) {

            return 0;

        }

        // Option 1: skip current node, directly rob its child node

        int skip = helper(node.left, false) + helper(node.right, false);

        // Option 2: rob current node (only when its parent node not robbed)

        int rob = 0;

        if(!parentRobbed) {

            rob = node.val + helper(node.left, true) + helper(node.right, true);

        }

        return Math.max(skip, rob);

    }

}

Time Complexity: O(2^n)

Space Complexity: O(n)

**Solution 2: Memoization (30 min)**

**Wrong Solution**

class Solution {

    public int rob(TreeNode root) {

        Map<TreeNode, Integer> memo = new HashMap<>();

        return helper(root, false, memo);

    }

    private int helper(TreeNode node, boolean parentRobbed, Map<TreeNode, Integer> memo) {

        if(node == null) {

            return 0;

        }

        if(memo.containsKey(node)) {

            return memo.get(node);

        }

        // Option 1: skip current node, directly rob its child node

        int skip = helper(node.left, false, memo) + helper(node.right, false, memo);

        // Option 2: rob current node (only when its parent node not robbed)

        int rob = 0;

        if(!parentRobbed) {

            rob = node.val + helper(node.left, true, memo) + helper(node.right, true, memo);

        }

        int result = Math.max(skip, rob);

        memo.put(node, result);

        return result;

    }

}

**Correct Solution by taking account for boolean flag 'parentRobbed'**

class Solution {

    public int rob(TreeNode root) {

        Map<Pair<TreeNode, Boolean>, Integer> memo = new HashMap<>();

        return helper(root, false, memo);

    }

    private int helper(TreeNode node, boolean parentRobbed, Map<Pair<TreeNode, Boolean>, Integer> memo) {

        if(node == null) return 0;

        Pair<TreeNode, Boolean> key = new Pair<>(node, parentRobbed);

        if (memo.containsKey(key)) {

            return memo.get(key);

        }

        int skip = helper(node.left, false, memo) + helper(node.right, false, memo);

        int rob = 0;

        if(!parentRobbed) {

            rob = node.val + helper(node.left, true, memo) + helper(node.right, true, memo);

        }

        int result = Math.max(skip, rob);

        memo.put(key, result);

        return result;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Solution 3: Another Memoization Solution**

class Solution {

    public int rob(TreeNode root) {

        int[] result = helper(root);

        return Math.max(result[0], result[1]);

    }

    // Instead of tracking parentRobbed, use the standard DP-on-trees

    // approach that returns both states (rob/skip) for each node

    private int[] helper(TreeNode node) {

        if(node == null) {

            // {rob, skip}

            return new int[2];

        }

        int[] left = helper(node.left);

        int[] right = helper(node.right);

        // robCurrent: node.val + left[1] (skip left) + right[1] (skip right)

        int rob = node.val + left[1] + right[1];

        // skipCurrent: max(left[0], left[1]) (skip or rob left) +

        // max(right[0], right[1]) (skip or rob right)

        int skip = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);

        return new int[]{rob, skip};

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to Deepseek**

**Can you write up a Native DFS solution ?**

Here's a **native DFS (brute-force) solution** for LeetCode 337. House Robber III, which explores all possible robbery combinations without optimization:

**Native DFS Solution (Brute-Force)**

class Solution {

public int rob(TreeNode root) {

return dfs(root, false);

}

private int dfs(TreeNode node, boolean parentRobbed) {

if (node == null) {

return 0;

}

// Option 1: Skip current house

int skip = dfs(node.left, false) + dfs(node.right, false);

// Option 2: Rob current house (only if parent wasn't robbed)

int rob = 0;

if (!parentRobbed) {

rob = node.val + dfs(node.left, true) + dfs(node.right, true);

}

return Math.max(skip, rob);

}

}

**Key Characteristics:**

**Unoptimized Recursion**:

Explores all valid robbery paths (rob/skip decisions) without memoization.

Time Complexity: **O(2^n)** in worst case (highly inefficient for large trees).

**Decision Logic**:

parentRobbed flag prevents robbing two adjacent nodes.

Each node has two choices:

**Skip**: Move to children with parentRobbed = false.

**Rob**: Add node's value and move to children with parentRobbed = true.

**Base Case**:

Returns 0 for null nodes (no houses to rob).

**Why This is Inefficient:**

**Exponential Redundancy**:

For a full binary tree:

- Calls duplicate subproblems repeatedly.

- Example: dfs(node.left.left) may be computed multiple times.

**No State Caching**:

Unlike the DP solution, it doesn't store/reuse intermediate results.

**Comparison with Optimized Solutions:**

|  |  |  |
| --- | --- | --- |
| Aspect | Native DFS | Optimized DP (Solution 3) |
| **Time Complexity** | O(2ⁿ) | O(n) |
| **Space Complexity** | O(n) recursion stack | O(n) recursion stack |
| **Redundancy** | High (repeats work) | None (memoized states) |

**When to Use This:**

**Educational purposes only** (to understand the problem's brute-force logic).

**Small trees** (n < 20).

**Never for LeetCode submissions** (will time out on larger inputs).

For actual solutions, always prefer the **DP approach with state tracking** (as shown in my previous answer). This native DFS is purely to demonstrate the foundational recursive logic.

**Why my memoization solution is wrong initially ?**

Your memoization solution is incorrect because it **doesn't account for the parentRobbed state in the memoization key**. Here's why it fails and how to fix it:

**Problem in Your Solution**

**Incomplete Memoization Key**:

You only use TreeNode as the memo key.

The result for a node **depends on whether its parent was robbed** (parentRobbed), but this isn't part of the key.

This causes incorrect cache hits when the same node is visited with different parentRobbed states.

**Example Failure**:

For the tree [3,2,3,null,3,null,1]:

Your code caches node=3 (val=3) with result=7 when parentRobbed=false.

Later, it incorrectly reuses this cached value when parentRobbed=true, leading to overcounting.

**Fixed Solution**

class Solution {

public int rob(TreeNode root) {

Map<Pair<TreeNode, Boolean>, Integer> memo = new HashMap<>();

return helper(root, false, memo);

}

private int helper(TreeNode node, boolean parentRobbed, Map<Pair<TreeNode, Boolean>, Integer> memo) {

if (node == null) return 0;

Pair<TreeNode, Boolean> key = new Pair<>(node, parentRobbed);

if (memo.containsKey(key)) {

return memo.get(key);

}

int skip = helper(node.left, false, memo) + helper(node.right, false, memo);

int rob = 0;

if (!parentRobbed) {

rob = node.val + helper(node.left, true, memo) + helper(node.right, true, memo);

}

int result = Math.max(skip, rob);

memo.put(key, result);

return result;

}

}

**Key Fixes**

**Enhanced Memo Key**:

Now uses Pair<TreeNode, Boolean> to track both the node and parentRobbed state.

This ensures unique cache entries for different robbery states.

**Correct State Tracking**:

When parentRobbed=true, the solution correctly skips the current node.

No more incorrect cache reuse.

**And below is another Memozation Solution**

Here's a **memoization solution** built directly from the native DFS approach, using a HashMap to cache results and avoid redundant calculations:

class Solution {

public int rob(TreeNode root) {

// Memoization map: Key = TreeNode, Value = Pair of (rob, skip)

Map<TreeNode, int[]> memo = new HashMap<>();

int[] result = dfs(root, memo);

return Math.max(result[0], result[1]);

}

private int[] dfs(TreeNode node, Map<TreeNode, int[]> memo) {

if (node == null) {

return new int[]{0, 0};

}

// Return cached result if available

if (memo.containsKey(node)) {

return memo.get(node);

}

int[] left = dfs(node.left, memo);

int[] right = dfs(node.right, memo);

// robCurrent: node.val + left[1] (skip left) + right[1] (skip right)

int robCurrent = node.val + left[1] + right[1];

// skipCurrent: max(left[0], left[1]) + max(right[0], right[1])

int skipCurrent = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);

// Cache and return the result

int[] res = new int[]{robCurrent, skipCurrent};

memo.put(node, res);

return res;

}

}

**Key Improvements Over Native DFS:**

**Memoization**:

Stores results for each node as [robCurrent, skipCurrent] to avoid recomputation.

Uses TreeNode as the key in the HashMap.

**State Tracking**:

Returns both possibilities (rob/skip) for each node, enabling efficient reuse.

**Performance**:

**Time Complexity**: O(n) – Each node is processed once.

**Space Complexity**: O(n) – For the memoization map and recursion stack.

**How It Works:**

**Base Case**: Returns [0, 0] for null nodes.

**Cache Check**: Returns precomputed results if available.

**Recursive Calculation**:

Computes robCurrent and skipCurrent using child states.

**Cache Update**: Stores results before returning.

**Comparison with Native DFS:**

|  |  |  |
| --- | --- | --- |
| Operation | Native DFS (O(2^n)) | Memoization (O(n)) |
| Compute  rob(3) | Recalculates 4 times | Computes once, reuses 3x |
| Total Calls | Exponential | Linear |

This solution preserves the intuitive DFS logic while adding memoization to achieve optimal performance. It's the natural evolution of the brute-force approach!

**Refer to**

<https://leetcode.com/problems/house-robber-iii/solutions/79330/step-by-step-tackling-of-the-problem/>

**Step I -- Think naively**

At first glance, the problem exhibits the feature of "optimal substructure": if we want to rob maximum amount of money from current binary tree (rooted at root), we surely hope that we can do the same to its left and right subtrees.

So going along this line, let's define the function rob(root) which will return the maximum amount of money that we can rob for the binary tree rooted at root; the key now is to construct the solution to the original problem from solutions to its subproblems, i.e., how to get rob(root) from rob(root.left), rob(root.right), ... etc.

Apparently the analyses above suggest a recursive solution. And for recursion, it's always worthwhile figuring out the following two properties:

Termination condition: when do we know the answer to rob(root) without any calculation? Of course when the tree is empty ---- we've got nothing to rob so the amount of money is zero.

Recurrence relation: i.e., how to get rob(root) from rob(root.left), rob(root.right), ... etc. From the point of view of the tree root, there are only two scenarios at the end: root is robbed or is not. If it is, due to the constraint that "we cannot rob any two directly-linked houses", the next level of subtrees that are available would be the four "**grandchild-subtrees**" (root.left.left, root.left.right, root.right.left, root.right.right). However if root is not robbed, the next level of available subtrees would just be the two "**child-subtrees**" (root.left, root.right). We only need to choose the scenario which yields the larger amount of money.

Here is the program for the ideas above:

public int rob(TreeNode root) {

if (root == null) return 0;

int val = 0;

if (root.left != null) {

val += rob(root.left.left) + rob(root.left.right);

}

if (root.right != null) {

val += rob(root.right.left) + rob(root.right.right);

}

return Math.max(val + root.val, rob(root.left) + rob(root.right));

}

However the solution runs very slowly (1186 ms) and barely got accepted (the time complexity turns out to be exponential, see my [comments](https://discuss.leetcode.com/topic/39834/step-by-step-tackling-of-the-problem/26?page=2) below).

**Step II -- Think one step further**

In step I, we only considered the aspect of "optimal substructure", but think little about the possibilities of overlapping of the subproblems. For example, to obtain rob(root), we need rob(root.left), rob(root.right), rob(root.left.left), rob(root.left.right), rob(root.right.left), rob(root.right.right); but to get rob(root.left), we also need rob(root.left.left), rob(root.left.right), similarly for rob(root.right). The naive solution above computed these subproblems repeatedly, which resulted in bad time performance. Now if you recall the two conditions for dynamic programming (DP): "**optimal substructure**" + "**overlapping of subproblems**", we actually have a DP problem. A naive way to implement DP here is to use a hash map to record the results for visited subtrees.

And here is the improved solution:

public int rob(TreeNode root) {

return robSub(root, new HashMap<>());

}

private int robSub(TreeNode root, Map<TreeNode, Integer> map) {

if (root == null) return 0;

if (map.containsKey(root)) return map.get(root);

int val = 0;

if (root.left != null) {

val += robSub(root.left.left, map) + robSub(root.left.right, map);

}

if (root.right != null) {

val += robSub(root.right.left, map) + robSub(root.right.right, map);

}

val = Math.max(val + root.val, robSub(root.left, map) + robSub(root.right, map));

map.put(root, val);

return val;

}

The runtime is sharply reduced to 9 ms, at the expense of O(n) space cost (n is the total number of nodes; stack cost for recursion is not counted).

**Step III -- Think one step back**

In step I, we defined our problem as rob(root), which will yield the maximum amount of money that can be robbed of the binary tree rooted at root. This leads to the DP problem summarized in step II.

Now let's take one step back and ask why we have overlapping subproblems. If you trace all the way back to the beginning, you'll find the answer lies in the way how we have defined rob(root). As I mentioned, for each tree root, there are two scenarios: it is robbed or is not. rob(root) does not distinguish between these two cases, so "information is lost as the recursion goes deeper and deeper", which results in repeated subproblems.

If we were able to maintain the information about the two scenarios for each tree root, let's see how it plays out. Redefine rob(root) as a new function which will return an array of two elements, the first element of which denotes the maximum amount of money that can be robbed if root is **not robbed**, while the second element signifies the maximum amount of money robbed if it is **robbed**.

Let's relate rob(root) to rob(root.left) and rob(root.right)..., etc. For the 1st element of rob(root), we only need to sum up the larger elements of rob(root.left) and rob(root.right), respectively, since root is not robbed and we are free to rob its left and right subtrees. For the 2nd element of rob(root), however, we only need to add up the 1st elements of rob(root.left) and rob(root.right), respectively, plus the value robbed from root itself, since in this case it's guaranteed that we cannot rob the nodes of root.left and root.right.

As you can see, by keeping track of the information of both scenarios, we decoupled the subproblems and the solution essentially boiled down to a greedy one. Here is the program:

public int rob(TreeNode root) {

int[] res = robSub(root);

return Math.max(res[0], res[1]);

}

private int[] robSub(TreeNode root) {

if (root == null) return new int[2];

int[] left = robSub(root.left);

int[] right = robSub(root.right);

int[] res = new int[2];

res[0] = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);

res[1] = root.val + left[0] + right[0];

return res;

}

**How Solution 3 is more advantage than Solution 2 which a DP solution based on solution 1 ?**

You're absolutely right to compare these solutions! Let me break down the key differences and advantages between Solution 2 (DP with memoization) and Solution 3 (the optimized recursive approach similar to mine) from the LeetCode article:

**Key Comparison: Solution 2 vs. Solution 3**

|  |  |  |
| --- | --- | --- |
| Feature | Solution 2 (DP with Memoization) | Solution 3 (Optimized Recursive) |
| **Approach** | Top-down DP with HashMap memoization | Recursive with state propagation |
| **State Tracking** | Stores single max value per node | Tracks two states (rob/skip) per node |
| **Space Complexity** | O(n) (memo storage) | O(n) (recursion stack) |
| **Time Complexity** | O(n) (visits each node once) | O(n) (visits each node once) |
| **Code Complexity** | Requires explicit memo management | Cleaner recursive logic |
| **Overlap Handling** | Explicitly avoids recomputation via HashMap | Implicitly avoids recomputation via state propagation |

**Why Solution 3 is More Advantageous:**

**No Explicit Memoization**:

Solution 3 avoids the overhead of maintaining a HashMap by propagating both states (rob/skip) in one pass .

Example: When calculating for a node, it immediately gets the optimal sub-solutions from children without redundant calls.

**Intuitive State Representation**:

Returns [robCurrent, skipCurrent] pairs, making the decision logic clearer:

return new int[]{

root.val + left[1] + right[1], // Rob current

Math.max(left[0], left[1]) + Math.max(right[0], right[1]) // Skip current

};

This directly mirrors the problem's constraints .

**Better Constant Factors**:

Eliminates HashMap lookups/insertions, reducing runtime constants .

In practice, this often makes Solution 3 faster despite identical time complexity.

**Easier to Generalize**:

The two-state approach naturally extends to similar tree DP problems (e.g., binary tree coloring games) .

**When Solution 2 Might Be Preferable:**

If you need to query intermediate results (the HashMap acts as a cache).

In languages where recursive calls are expensive (though Java handles this well).

**Example Walkthrough**

For the tree:

3

/ \

2 3

\ \

3 1

**Solution 3** processes nodes in post-order:

Leaf nodes return [3,0] and [1,0].

Node 2 computes: [2+0+0, max(3,0)+max(0,0)] = [2,3].

Root 3 computes: [3+3+1, max(2,3)+max(3,1)] = [7,5].

Final result: max(7,5) = 7 .

**Conclusion**

Solution 3 is generally superior because:

It's more elegant (no manual memoization).

Has better runtime performance in practice.

More clearly expresses the problem's optimal substructure .

Both solutions are O(n) time/space, but Solution 3's design makes it the preferred choice for interviews and production code.

**Refer to**

[L198.House Robber (Ref.L213,L337,L2320,L2560)](note://WEB3a5c4401c63ae93266b0b603cb20076c)

[L213.House Robber II (Ref.L198,L337,L2320,L2560)](note://A5D6119BDEB54084A860F30A71A0A8C2)

[L2560.House Robber IV (Ref.L11,L198)](note://WEB4bc49b4b30992ee8ff1d6f950fa72a1f)