House Robber 3

Sunday, December 5, 2021 8:43 PM

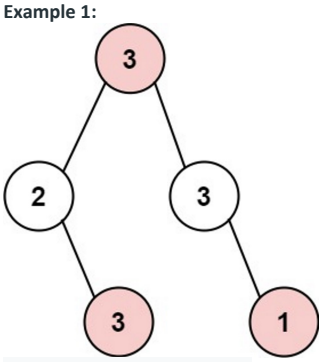
337. House Robber III

Medium

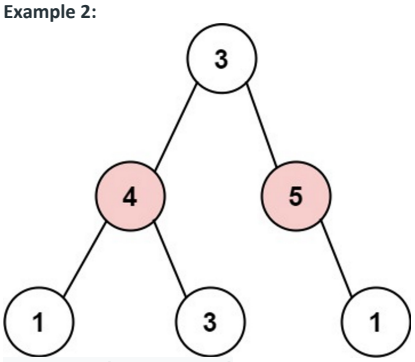
538586Add to ListShare

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, 104].
- 0 <= Node.val <= 104

Accepted
254,341
Submissions
480,645

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

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:

1. 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.

1. 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.

```
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 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.

```
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;
}
```

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