Solution: House Robber III

Let's solve the House Robber III problem using the Backtracking pattern.

We'll cover the following

- Statement
- Solution
  - Naive approach
  - Optimized solution using backtracking
    - Solution summary
    - Time complexity
    - Space complexity

Statement

A thief has discovered a new neighborhood to target, where the houses can be represented as nodes in a binary tree. The money in the house is the data of the respective node. The thief can enter the neighborhood from a house represented as `root` of the binary tree. Each house has only one parent house. The thief knows that if he robs two houses that are directly connected, the police will be notified. The thief wants to know the maximum amount of money he can steal from the houses without getting caught by the police. The thief needs your help determining the maximum amount of money he can rob without alerting the police.

Constraints:

- The number of nodes in the tree is in the range $[1, 10^4]$.
- $0 \leq$ `node.data` $\leq 10^4$

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow, based on considerations such as time complexity and any implementation constraints.

Naive approach

A naive approach to solving this problem involves calculating the sum of every possible valid combination of houses that can be robbed. A valid combination is one in which the thief does not rob two houses that are directly connected. After calculating all possible valid combinations, we can return the sum of the combination of houses that produces the maximum amount of money. This approach requires evaluating all 2^n possible ways to rob the neighborhood, which results in exponential time complexity. We can improve the algorithm's efficiency by using backtracking.

Optimized solution using backtracking

An optimized approach to solve this problem is using backtracking, which involves recursively exploring all possible combinations and backtracking whenever we encounter an invalid combination. By doing so, we can avoid evaluating redundant solutions and reach the optimal combination more efficiently.

Our solution aims to find the maximum amount of money a thief can rob from the houses in a binary tree without alerting the police. The `rob` function takes the `root` node of the binary tree as input, calls the `heist` function, and returns the maximum amount of money that the thief can rob from the binary tree.

The `heist` function recursively calculates the maximum amount of money that can be robbed from a subtree rooted at a node. It returns a pair containing the values of `includeRoot` and `excludeRoot`.

- `includeRoot` contains the maximum amount of money that can be robbed from the subtree rooted at the node with the `root` node included.
- `excludeRoot` contains the maximum amount of money that can be robbed from the subtree rooted at the node with the `root` node excluded.

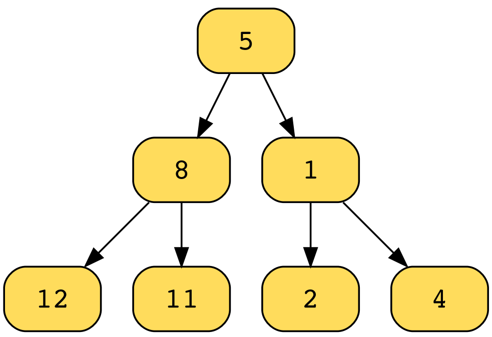
The function proceeds through the following steps:

- If the tree is empty, the heist function returns `[0, 0]`, which represents the `includeRoot` and `excludeRoot`, respectively.
- Otherwise, it recursively calculates the maximum amount of money that can be robbed from the left and right subtrees of the `root` node.
- Then, it stores the maximum amount of money that can be robbed from the `root` node in the `includeRoot`. It computes the maximum amount by adding the value of the `root` node to the following:
  - The maximum amount of money that can be robbed from the left subtree, excluding their respective parent node.
  - The maximum amount of money that can be robbed from the right subtree, excluding their respective parent node.
- Alternatively, it computes the sum of the maximum amount of money that can be robbed from both the left subtree and right subtree of the `root` node, excluding the `root` node itself, and store it in the `excludeRoot`.
- Finally, the `heist` function returns a pair containing the values of `includeRoot` and `excludeRoot`.

The `rob` function then returns the maximum value from the pair obtained by the `heist` function, which is the maximum amount of money the thief can rob from the houses in a binary tree without alerting the police.

Let's look at the following illustration to get a better understanding of the solution:

We have the amount of money that we can rob, arranged in this binary tree. We will use the algorithm above and recursively calculate the amount that can be robbed from the left and right subtrees of the **root** node.



1 of 12

Let's implement the algorithm as discussed above:

Java

Main.java

BinaryTree.java

TreeNode.java

```
public class Main {
    public static int[] heist(TreeNode<Integer> root) {
        // Empty tree case
        if (root == null) {
            return new int[]{0,0};
        }
        // Recursively calculating the maximum amount that can be robbed fron
        int[] leftSubtree = heist(root.left);
        // Recursively calculating the maximum amount that can be robbed fron
        int[] rightSubtree = heist(root.right);
        // includeRoot contains the maximum amount of money that can be robbe
        int includeRoot = root.data + leftSubtree[1] + rightSubtree[1];
        // excludeRoot contains the maximum amount of money that can be robbe
        int excludeRoot = Math.max(leftSubtree[0], leftSubtree[1]) + Math.ma›

        return new int[] {includeRoot, excludeRoot};
    }

    public static int rob(TreeNode<Integer> root) {
       // Returns maximum value from the pair: [includeRoot, excludeRoot]
        int[] result = heist(root);
        return Math.max(result[0], result[1]);
    }

    public static void main(String[] args) {
```

House Robber III

Solution summary

1. If the tree is empty, return 0.
2. Otherwise, recursively calculate the maximum amount of money that can be robbed from the left and right subtrees of the `root` node.
3. Compute the maximum amount of money that can be robbed with the parent node included.
4. Compute the maximum amount of money that can be robbed with the parent node excluded.
5. Return the maximum value from the following:
    - The maximum amount of money that can be robbed with the parent node included.
    - The maximum amount of money that can be robbed with the parent node excluded.

Time complexity

The time complexity of this solution is $O(n)$, where n is the number of nodes in the tree, since we visit all nodes once.

Space complexity

The space complexity of this solution is $O(n)$, since the maximum depth of the recursive call tree is the height of the tree, which is n in the worst case, and each call stores its data on the stack.

← Back

House Robber III

Next →

Restore IP Addresses

Mark as Completed